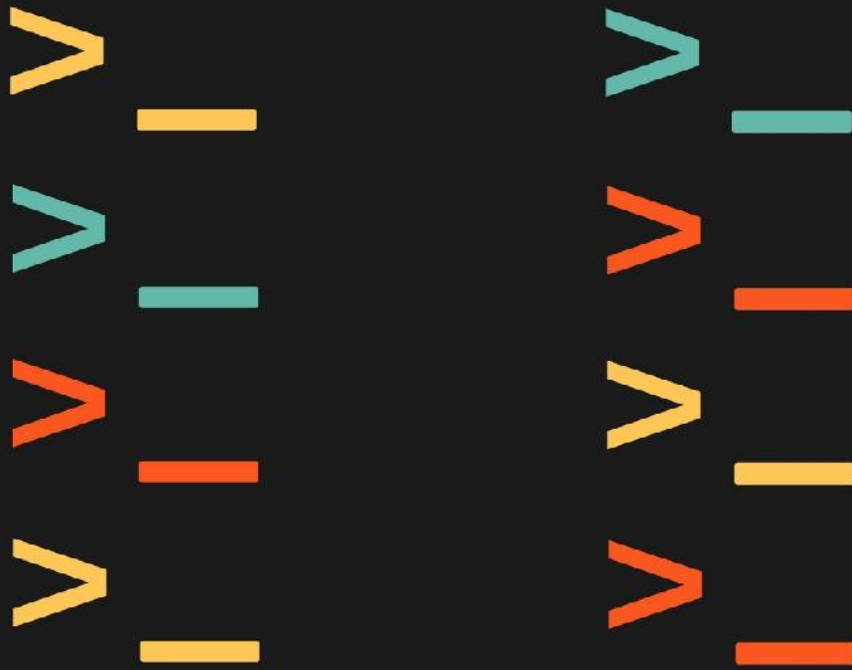


Foreword by Eric Holscher



DOCS LIKE CODE

Write, review, test, merge, build, deploy, repeat.

BY ANNE GENTLE

Table of Contents

Docs Like Code

Copyright

Foreword

About this book

INTRODUCING DOCS LIKE CODE

Why treat docs like code?

PLAN FOR DOCS LIKE CODE

Specialty information: REST API docs

What about wikis?

OPTIMIZE DOCS-LIKE-CODE WORKFLOWS

Author and build content

Automate builds

Review docs

Publish docs 1.7.4

TUTORIAL: GET STARTED WITH DOCS LIKE
CODE

LESSONS LEARNED WITH DOCS LIKE CODE

Glossary: Understand Git concepts

Docs Like Code

Write, review, test, merge, build, deploy, repeat.

By Anne Gentle

Copyright

Copyright © 2017 Anne Gentle



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.

Apache, jclouds, and Maven are registered trademarks of the Apache Software Foundation.

Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries.

GitHub is a trademark registered in the United States by GitHub, Inc.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

The OpenStack Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and

other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Python is a registered trademark of the Python Software Foundation.

Rackspace is a registered trademark of Rackspace US, Inc.

Red Hat and OpenShift are trademarks of Red Hat, Inc., registered in the U.S. and other countries.

Any use of a trademarked name without a trademark symbol is for readability purposes only. We have no intention of infringing on the trademark.

Disclaimer

The information in this book is provided without warranty.

The author, contributors, and publisher have neither liability nor responsibility to any person or entity related to any loss or damages arising from the information contained in this book.

Second edition: 2017

ISBN 978-1-387-07512-6

Just Write Click

Austin, TX, USA

<http://docslikecode.com>

Foreword

Hey there, thanks for your interest in the docs-like-code world. I'd like to be the first to welcome you, and give you a taste of what's to come.

Throughout my career as a developer, I've seen the wonders that come from getting documentation and code in the same workflow. My first experience with this was cofounding [Read the Docs](#), which is a service that hosts open source software documentation. Its audience is primarily developers, and by bringing docs inside the development workflow, developers have embraced documentation at a massive scale. In 2016 alone, we served [250 million page views](#) of documentation, almost completely written by developers of open source projects. During 2016, the number of projects using Read the Docs grew from 28,000 to 53,673, which reflects over 90% growth year-over-year.

While Read the Docs has been successful at getting open source developers to write documentation, I feel that documentation still plays second fiddle to the code. So in 2013 I helped cofound [Write the Docs](#), which advocates for the value of documentation in the software industry. The community there has taught me many things, one of the biggest being that technical writers get more collaboration and ownership from their product teams in a docs-like-code workflow.

At our 2015 US conference, [Riona MacNamara](#) talked about how adopting docs like code has completely transformed how Google does documentation. At our 2016 US conference, a [panel](#) with folks from Rackspace, Microsoft, Balsamiq, and Twitter talked about adopting these practices. The docs-like-code concepts are widely practiced in the industry, and each conference provides more inspiring examples.

This book is a practical introduction to the docs-like-code mindset. It gives you an overview of the tools and processes that allow you to integrate it into your team. From making small adjusts in workflows, to transforming how your team works, you will get value out of the information in this book.

Anne, Diane, and Kelly have worked on one of the largest open source code bases at OpenStack and a documentation transformation at Rackspace. They distill their experiences into actionable advice, along with case studies that show how it works in practice. The book includes a wonderful tutorial that walks you through your first project, as well as hard-won advice that comes from years of putting these ideas into practice.

The docs-like-code concepts are still evolving and being refined. This book is the best introduction to them that exists. I hope that you enjoy reading it, and are able to introduce the practices described into your work. I truly believe that treating docs like code is the future of software documentation, and I'm glad you're along for the ride :)

Eric Holscher

Cofounder of Read the Docs & Write the Docs

Portland, Oregon

Jan 30, 2017

About this book

- [Author Anne Gentle](#)
- [Contributor Diane Fleming](#)
- [Contributor Kelly Holcomb](#)
- [How this book was created](#)

Author Anne Gentle

Anne Gentle works as a technical product manager at Cisco, supporting the Metacloud™ product based on OpenStack®. She uses open-source techniques for API design, documentation, dev ops, and developer support. OpenStack provides open-source cloud computing software through collaboration among more than 30 projects written in Python across hundreds of Git™ repositories. Anne advocates for cloud users and administrators by providing accurate technical information to increase OpenStack adoption as a cloud for the world.

Before working on OpenStack in 2010, Anne worked as a community publishing consultant, providing strategic direction for professional writers who wanted to produce online content in collaborative ways. Her enthusiasm for community-based documentation methods prompted her to write a book about using social publishing techniques for technical documentation titled *Conversation and Community: The Social Web for Documentation*.

As a supporter of women in technology, Anne coordinated efforts to connect OpenStack to the GNOME Outreach Program for Women, now named Outreachy. She also has an interest in health data in the cloud, using her cloud knowledge to monitor her type I diabetic son through a continuous glucose monitoring system. She writes about various

techniques for web content development at [Just Write Click](#) and [Docs Like Code](#). As the mom of two youngsters with an awesome husband, she relishes every opportunity to talk to people who are as fascinated with technology as she is.

Contributor Diane Fleming

Diane Fleming is one of the main REST API docs writers and an enthusiastic GitHub user at [PayPal](#). She started her career at IBM when both tech writers and programmers had to attend the lengthy in-house programming school. After her immersion in Assembler language and data dumps, she started as a tech writer for mainframes, spent several years as a programmer, and returned to tech writing at companies such as BMC Software, Vignette, Informatica, Rackspace, and Cisco.

She cut her GitHub teeth by working at Rackspace on REST API docs for both Rackspace products and OpenStack services. At first, the switch from more traditional, pessimistic locking systems such as Perforce to the optimistic GitHub system was a challenge for her. The paradigm shift from *owning books* to *contributing content* was hard-won. But now, she looks forward to the interactions that occur in a community where everyone is encouraged to contribute.

Contributor Kelly Holcomb

Kelly Holcomb is the Senior Technical Editor at Rackspace, working with multiple writing and development teams using GitHub for docs. Although she's been correcting people's grammar since middle school, high-tech companies such as Rackspace and BMC Software have been paying her to do it for almost 20 years. As a technical editor, she loves to take any piece of writing and make it better. And although she shed a few, hot tears while learning GitHub, she's growing to appreciate the collaborative environment that it provides.

How this book was created

Anne, Diane, and Kelly worked at Rackspace, each in a different role, but all making similar observations and working toward transforming how information is written, produced, and edited.

With Anne as the chief cook and bottle-washer (and instigator), we wrote this book together to chronicle the docs-like-code way of working. We maintained the book's source files in a private GitHub repo, authored the source files in Markdown, and configured the repo to use [GitBook](#) for automatic builds of PDF, EPUB, and HTML output.

Diane and Anne wrote furiously for a few months, borrowing from their experiences of using and teaching others to use these techniques and from Anne's articles that describe how [OpenStack docs](#) use these techniques.

Kelly first edited the chapters individually by creating Word documents from the Markdown files, marking them up, and creating PDFs from them to make the tracked changes easier to read. Then, she created issues for each chapter's edits, and Diane and Anne made pull requests with the suggested edits. As a final quality check, Kelly edited the book as a whole, creating pull requests with final copy edits.

Made from equal parts inspiration and perspiration, we hope that this book introduces your team to the docs-like-code techniques and ultimately improves your team's productivity

and the quality of its docs.

INTRODUCING DOCS LIKE CODE

We know you.

You love to write, and you're proud of your docs. But you want to change the way that you think about those docs and modernize how you make them. Where do you start?

Do what we do.

We use Git, a distributed version control system, and GitHub, a social coding site, for docs. Whether we're writing, reviewing, or deploying, we love to treat *docs like code*.

We've been using docs-like-code techniques for multiple projects in many contexts, from open-source software docs to enterprise doc sites. We believe in collaborative writing and reviewing, and in using development tools for docs to bring dev and docs closer together and to push the edges of imaginary boundaries.

You need a quick win to prove to yourself and your team that you're ready for a new mindset, toolset, and doc set.

Let's go.

Why treat docs like code?

Picture yourself writing in isolation. You have a vague understanding of the product you're writing about and who you're writing for. And you aren't completely sure what happens when you throw your content over the imaginary wall to the product developers and users.

You want user-centered docs but instead you get project-centered docs. You want technically accurate docs but instead you get vague hand waving from your reviewers and a dearth of tests. You want to focus on content but the complexities of the build system mean you babysit builds instead of writing and testing new tutorials.

What about combining the power of developer tools and techniques with the discipline and advocacy of writing excellent documentation? Why not treat docs like software code?

- [Defining *docs like code*](#)
- [Reasons why we treat docs like code](#)
- [Goals of docs like code](#)
- [Summary](#)

Defining *docs like code*

Over the years, docs have changed along with the technologies that they document. At their start, tech docs included formal specifications, assembly diagrams, and glossaries of technical terms. These docs targeted a literate audience with a manufacturing or mechanical background. Now, software docs are developed by using Agile techniques, and the docs themselves include interactive coding examples. But whether tech docs explain sewing machines or a REST API, their main goal has always been to aid in the effective use of technology.

When we say *docs*, we mean streamlined, tightly phrased, and fast-moving information that helps developers understand complex application interfaces. Docs can be anything from a single web page for a startup to an entire developer reference site. Modern docs, with their web and mobile interfaces and supportive user experience, are purposeful, instructive, and even beautiful.

When we say *treat docs like code*, we mean that you:

- Store the doc source files in a version control system.
- Build the doc artifacts automatically.
- Ensure that a trusted set of reviewers meticulously reviews the docs.
- Publish the artifacts without much human intervention.

These techniques constitute a docs-like-code framework.

Ideally, you seek ways to fit the docs into the development systems that are already in place. We have used [GitHub](#), [Gerrit](#), [Bitbucket®](#), [Git](#), and [GitLab](#), and other code systems to treat docs like code. This book leans heavily on our experiences with Git, GitHub, and Gerrit, but we do not intend to describe a one-size-fits all solution here. Our intent is to show how these tools provide a way to think about including docs output with software and services, and how to leverage continuous integration and review systems while writing collaboratively. Read this book, get inspired, and then work with your teams to treat docs like code together.

Reasons why we treat docs like code

Coders and writers have borrowed tools and techniques from each other for decades. When you look at the history of producing docs and code together, you see that tools were invented specifically to produce docs while coding.

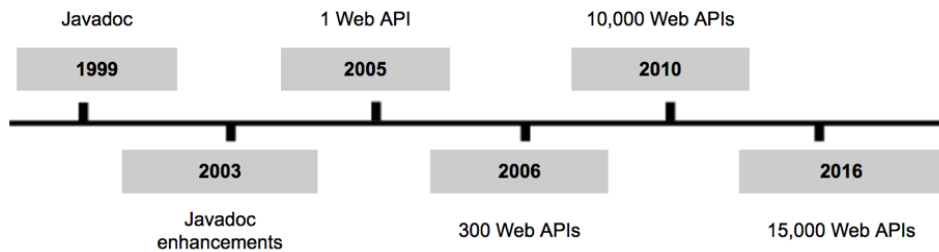
For example, the JavaDoc tool has been available since the first Java release in 1999. Visualizing the code in HTML, providing accessible online docs, and updating the documentation with the code were all keys to its success.

Visualizing the code in HTML

What I mostly use for visualization is JavaDoc, because it's an abstract view of the public interface. I generate JavaDoc a lot as I'm designing and developing. I look at the HTML pages generated by JavaDoc and think, well, this looks kind of confusing. And I go back and make some changes to the code. I may not be able to see that it is confusing just by looking at the code. So I think having different ways to visualize code and designs as it is being developed can help guide the design.

—Bill Venners, [Visualizing with JavaDoc](#)

Additionally, the [ProgrammableWeb Research Center](#) shows exponential web API growth from 2005 to 2013, as illustrated in part of the following image, and these APIs need docs to succeed.



Many forces, such as efficiency, profit, and user experience, drive organizations to integrate docs with code. Let's look at these forces:

- [Workplace evolution](#)
- [Technology changes](#)
- [Market and business needs](#)

Workplace evolution

Today's technology teams are distributed across offices and countries, buildings and time zones, and everyone must communicate asynchronously with distributed teammates. GitHub and code conventions provide standards that make distributed work easier and more efficient to do, and notifications provide collaboration advantages while letting individual contributors control their own settings.

Try to remember the last time you worked on a company-issued desktop computer. The workforce is mobile, and our hardware has become more portable, so the docs have also

become mobile and cross-platform compatible.

Technology changes

The rise of the REST API, where web services provide application programming interfaces for many tasks, has been meteoric over the last six years. REST (Representational State Transfer) is an efficient and lightweight way to get data and perform actions on a service, using HTTP as the underlying protocol. With the increasing use of software as a service (SaaS) and the need to document both application interfaces and user interfaces, excellent docs are in demand.

Webhooks, or ways to trigger an automatic action, have played a large part in the successful integration of GitHub with other tools such as Slack, Jenkins, and email. Because webhooks send an HTTP payload to a preconfigured URL, GitHub can be used to do a multitude of tasks from builds to backups to deployments.

Market and business needs

The technology market embraces the ideas of open collaboration and open source, where the software source code is made available to be redistributed and modified within licensing parameters. Both the business and technology worlds have found a need for the models that go with open-source software. Increasingly, companies rely on open-source software to run their business. Based on the [Future of Open Source Survey from 2016](#) with over 1,300 responses, 78

percent of respondents say their company relies on open source. The number responding with an affirmative answer about reliance has doubled since 2010.

Businesses are always looking for cost efficiencies, whether in tooling or in headcount. The tooling for treating docs like code can be cost-effective, even with supportability factored in. Leveraging the community is not a cost-saving measure necessarily, but it can provide resources that are otherwise unavailable. In many markets, senior developers make more than senior tech writers, so if you can gain efficiencies by having them work together, you can save money. Sometimes the number of web services and APIs that must be documented simply can't be covered by a central tech writing team, so distributing the work across development, test, and docs teams makes sense for large-scale endeavors. For example, imagine going from 30 APIs to 500 APIs. Your teams simply must find a way to work together on the docs and distribute both responsibility and accountability.

Goals of docs like code

When you treat docs like code, you accomplish the following goals:

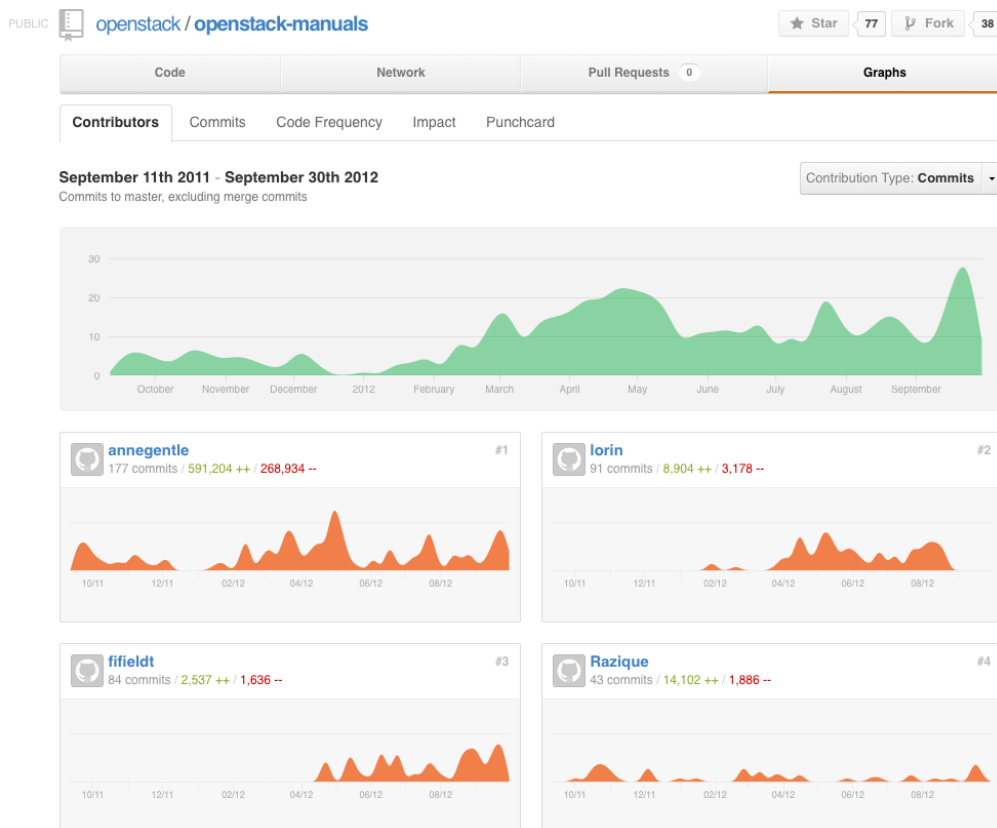
Promote collaboration	Collaborate with contributors efficiently by keeping docs in the same system as code with deliverables generated from source files.
Get long-tail contributions	Split deliverables into parts that encourage small but mighty contributions. One person no longer needs to own an entire deliverable.
Track doc bugs like code bugs	When you fix a doc bug, you reference that bug in the commit message to help reviewers judge whether the doc fix solves the stated problem.
Get better reviews	Trust team members to value docs, ensure technical accuracy and consistency, respect end users' needs, and advocate for the best doc deliverables for consumers.
Make beautiful docs	Learn enough about web development to be dangerous and create beautiful, modern docs.
Use developer tools and workflows	Apply software-development tools and techniques to software, API, and other technical docs. This includes automated builds that let you and your teams focus on content.
Get value from cost-effective tools	Use pay-as-you-need, flexible subscriptions to purchase the right level of tool help. Also, authoring tools are often free or inexpensive per-person.

Promote collaboration

By writing with and for your audience, you get to know your readers better. When you work with your readers in GitHub, you collaborate with your readers where they are.

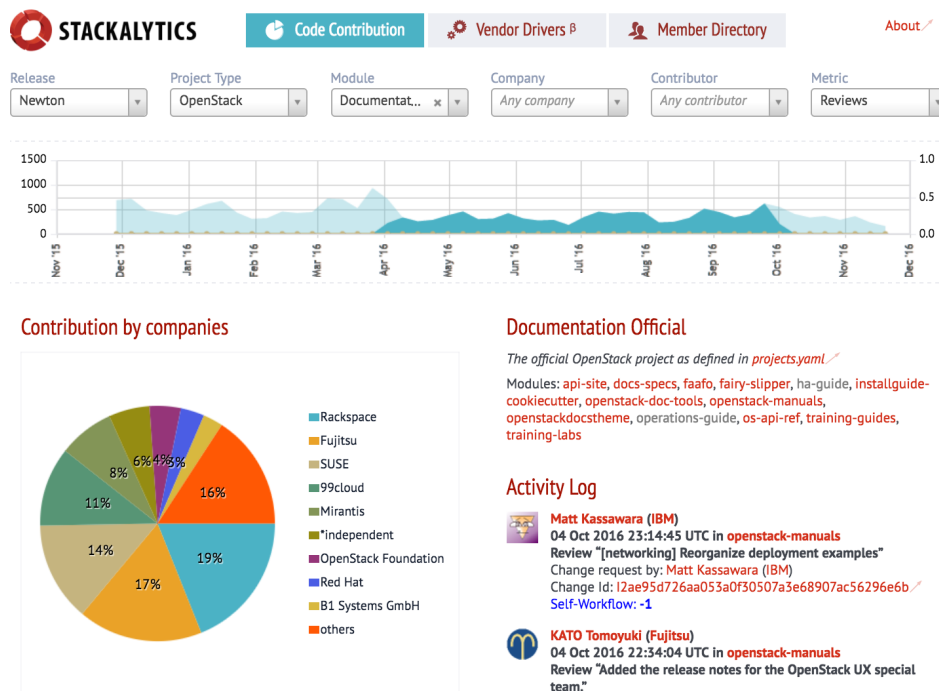
One way that GitHub in particular promotes collaboration is through contribution graphs, which let writers and coders see who works in which areas. Contributors' reputations are laid out in green blocks in the graphs. You can use the information in the graphs to recognize and reward contributors and to recruit the best talent. Contributors can use the graphs as incentive to build a better online resume by GitHub metrics.

Treating docs like code is *not* about devaluing writing by outsourcing it to a ragtag bunch of contributors for free. GitHub, for example, counts doc and dev contributions equally; doc contributors are valued as highly as code and test contributors. Your collaborators have a shared purpose in making great docs. People often like to “pay it forward,” reciprocating any support that they received, by writing helpful information for the docs.



GitHub and similar code systems avoid *documentation ghettos* because writers use the same tools that developers use. By adopting software techniques such as continuous integration, automated testing, and incremental improvement for docs, you get docs that are on par with the code itself. Also, you gain respect from those who value technical accuracy above all else, such as the app developers who consume your API docs. Keeping docs and code together also saves time and keeps context-switching to a minimum.

Sometimes, developers perceive that writers are distant from the product, and they consequently treat docs team members like second-class citizens. Written contributions, when treated like code, have the same value as developer code. We can show the metrics side-by-side.



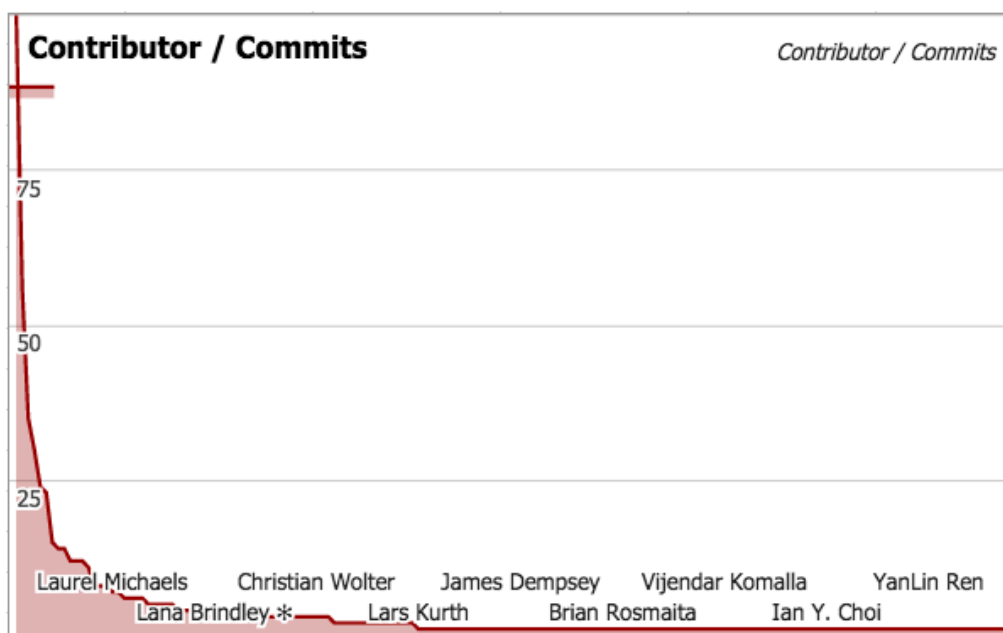
For example, in OpenStack, the patch and review processes are identical for code and docs, so the data used for event passes and voting privileges is the same for both developer contributions and technical writer contributions.

Get long-tail contributions

Chris Anderson, editor-in-chief at Wired Magazine, popularized the term *long tail* to refer to the phenomenon of the rise of the niche. For example, Amazon and Netflix earn a larger percentage of their profits from rentals and purchases of relatively obscure, niche books and movies (low-frequency events) than from rentals and purchases of best sellers and blockbusters (high-frequency events). This same model can be applied to technical docs: the niche, or long tail, docs are constructed from obscure knowledge gathering and hard-to-find information.

When you look for long-tail contributions, think of complex software products that only a few people understand deeply. If those people are motivated to write content for that small, focused area of the product, it makes those product docs better. Writing collaboratively helps you find like-minded people who are interested in your content.

Because a single writer can't know an entire software product, you must find specialists to collaborate with you. Those technical details can make a huge difference in the success of an information product, especially one targeted to developers. We recommend that you split deliverables into parts that encourage small but mighty contributions.



Track doc bugs like code bugs

If you don't know what's wrong with your doc, you can't fix it. By treating docs like code and logging and triaging issues, you can make a plan to work through that long backlog of the

broken parts of your docs and show incremental improvement over time.

Doc bugs can be tasks, outright errors, simple typos that make your docs look sloppy, or feature requests for the docs themselves. When you give your readers a way to track doc issues, they can tell you where they found confusing or incorrect content and even how to fix it.

When bug tracking gets the job done

Distributing doc bugs with collaborators around the globe in multiple time zones has its benefits. In OpenStack, Anne once logged a doc bug at the end of her workday, so that she could track it and remember to pick it up later.

By the time she woke up and sat at her desk for work the next day, instead of having to fix the bug herself, she simply had to review a patch that fixed it from a contributor on the other side of the globe, who was working during Anne's night hours.

Get better reviews

You might have seen review systems for enterprise docs in which the workflow simply moves frozen-in-time PDF files around for mindless sign-offs. When you treat docs like code, reviewers can see final changes in context before approving them.

Because code system workflows encourage small changes that are easy to compare, you can get more reviews. Also, you can encourage more interest in reviewing the docs by being in the notification system already used by developers on your team.

Working in the same collaboration tools as technical people enables better reviews for technical content. In OpenStack, as many as 50 patches per day can be merged by running four automated tests per patch and requiring two humans to review each patch and click a button to publish. Each reviewer who can publish must adhere to established review rules.

Reviewers who become contributors

Moving reviews to systems like GitHub also encourages incremental improvement, even if it takes more than three years. Anne once wrote an installation guide by herself, with limited knowledge, by interviewing one subject matter expert and repeatedly testing the instructions. It was a valiant effort, but not as real-world as OpenStack needed.

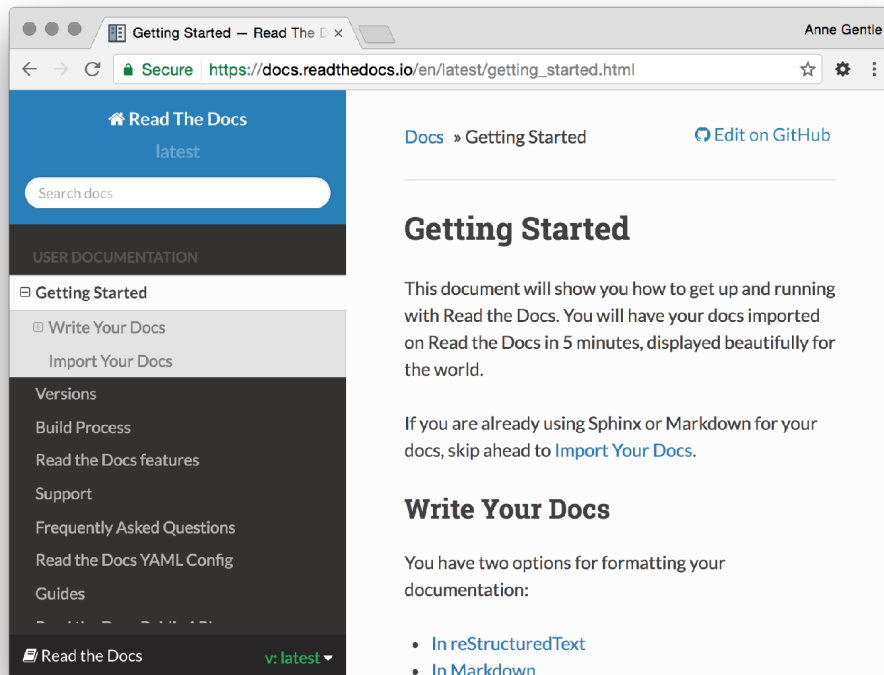
After reading that guide, another person completely tested the content as a proof-of-concept in a lab at his job. He pointed out many places where it could be reorganized, reshaped, and improved. Because his context was much more like that of the real audience, the guide was reshaped through his tests, reviews, and eventually his contributions.

Make beautiful docs

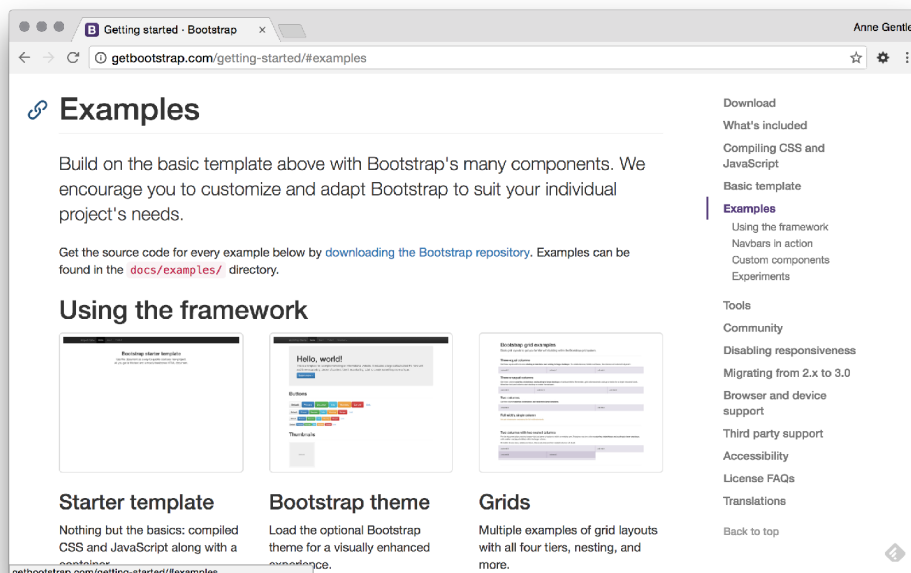
By treating docs like code, you enter a community that truly wants beautiful docs sites. We want widely accepted tooling that is built in the open and shared by developers and writers alike.

Repositories like Mark Phillips' [Beautiful Docs](#) collect examples of "...useful, well-written, and otherwise beautiful documentation." In the API docs world, there are many examples of good design and admiration for other sites designed in similar ways. Many API docs and testing tool companies have been acquired in the last two years, which highlights the value of attractive, useful doc design when the business value proposition of documentation is considered.

Great docs themes are clean, useful, and helpful in organizing the docs. The Read the Docs theme is often referred to as a "gold standard." The theme is responsive for mobile devices, has an expandable table of contents, and has a best-in-class version implementation. The [source code for the Sphinx theme](#) is available on GitHub.



Many themes use Bootstrap, which is an HTML, CSS, and JS framework. The [GetBootstrap.com](https://getbootstrap.com) site is another example of a clean documentation theme.



Use developer tools and workflows

Development teams perform continuous integration and deployment, completely mimicking their local environment in production and continuously updating the code and product. Docs can work the same way, and the tools that developers use can be applied in the docs context. Docs can be published automatically after trusted reviewers merge changes into the repository. Docs can have simple tests to ensure quality, such as white-space tests or spell checking. Docs can use continuous builds to ensure that they build and publish correctly so contributors can focus on content rather than on the build itself.

You can also create efficiencies for reference information by scraping the current code and using edited code content itself to write the docs. When you can document over 6,000 configuration options this way, you can focus efforts elsewhere on helpful docs and let the robots take care of the mounds of reference descriptions.

Read the Docs describes this goal so well on their website:

Read the Docs hosts docs, making it fully searchable and easy to find. You can import your docs using any major version control system, including Mercurial, Git, Subversion, and Bazaar. We support webhooks so your docs get built when you commit code. Version control enables you to build docs from tags and branches of your code in your repository.

Imagine if a restaurant chef never taught anyone how to prepare his most popular dish because he was the only person with the proper knife for the technique. Compare that scene to closely guarding secrets of proprietary docs tool chains with expensive per-seat licenses. Although we all know that certain jobs require specialty knowledge or tools, we shouldn't guard writing knowledge or tools in order to protect jobs.

Conversely, developer workflows shouldn't be mysterious to writers. Learning techniques while working side-by-side is a great way to demystify developer tools.

And from another perspective, developers shouldn't arrogantly think their code is all that matters, and docs don't matter because their users can "read the code." We can all think of examples of failures caused by a lack of empathy.

By treating docs like code, you open up possibilities for automation, continuous integration, deployment to web servers, and lights-out docs service for around-the-clock support. The software tools have changed, and the docs tools should change accordingly. In today's technology world, speed matters, and docs must keep up.

Get value from cost-effective tools

GitHub is a web-based user interface for collaboration and version control with [Git](#). GitHub works as a service, with a free tier for open source, public projects. [GitHub monthly subscriptions](#) range from \$7 per month for unlimited private repositories for an individual, to \$9 per month per user for an

organization, to \$21 per month for Enterprise GitHub, which you host on your company servers with your existing security controls.

Other code-hosting services include:

- The [GitLab](#) community edition, which is an open-source server. It has many of the same UI features as GitHub, but typically a company self-hosts GitLab. An enterprise edition is also available.
- [Bitbucket](#), which is part of the [Atlassian®](#) family of software tracking and development products, including the [Confluence®](#) wiki tool and the [JIRA®](#) bug tracker.

In addition to using the code-hosting and review tools offered by GitHub and other code-hosting services, you can gain efficiency by automating tests and builds. Many of the integrated build services that work with GitHub or BitBucket have a free tier for public repos, with monthly subscriptions for additional needs such as builds for private repos. When you want to automate testing, you should compare pricing based on the number of jobs, repositories, collaborators, and security considerations, including whether the repo is public or private. For example, [monthly plans for the Travis CI](#) continuous integration tool give you 100 builds for free to start, and then for one concurrent job on a public or private repo, you pay \$69 per month. The prices for other monthly plans range from \$129 per month for two jobs to \$489 per month for ten concurrent jobs.

Hosting the docs website might have associated costs, such as about \$10 per year for maintaining a custom domain name through a DNS registrar. You might need to pay for web hosting itself. If you want HTTPS, you can purchase an SSL certificate for about \$150 to \$640 per year depending on the number of subdomains and possible warranty ranges, or you can get free certificates through [Let's Encrypt](#) or other provisioning services that provide automation and free certificates. Using GitHub Pages for hosting has no associated cost, but you also can't use an SSL certificate unless you use a third-party cloud hosting system in addition to GitHub Pages. Bitbucket offers a similar hosting service through [Bitbucket Cloud](#) and uses HTTPS; however, it doesn't currently offer custom domains.

Lastly, the authoring tools for simple ASCII-based markup languages are often free or inexpensive. These writing tools are priced based on a per-user license, or they are completely free and open source. Writers and developers alike appreciate using command-line editors that can be readily configured. Side-by-side editors, in which you see the simple markup in one window and a rendered version in another window, cost about \$5 to \$70 per user for Mac [iA Writer](#) and [Sublime Text](#), respectively.

What about wikis?

Though wikis don't meet all the criteria of a docs-like-code framework, they do provide a simple and collaborative authoring environment with a low barrier to entry. See [What about wikis?](#)

Summary

Even if only some of these reasons for treating docs like code ring true in your situation, you can find parts that work well for your team and your deliverables. When you want to integrate with development, collaborate with technical users and developers, and create beautiful deliverables quickly and accurately, these techniques are for you.

PLAN FOR DOCS LIKE CODE

Many teams build a simple website from a single GitHub repo. But as your content outgrows this one-to-one model, you must organize your deliverables in a meaningful way while maintaining an architecture that maximizes your ability to send review notifications and automate builds.

To help contributors form a mental map of the docs, synchronize the way that you organize source files with the organization of your site. If you have a simple product, a single repo that contains both the code and docs can be easiest. For a complex set of products or projects, you have more choices about how you organize source files and provide navigation on your site.

- [Source file organization](#)
- [Inclusion mechanisms](#)
- [Use case: OpenStack docs](#)
- [Summary](#)

Source file organization

How do you know when to make a single deliverable from one repo or make a large website from multiple repos? The way that you organize information influences how you can present it to readers.

To decide whether to keep docs in the dev repo, use a single docs repo, or use multiple docs repos, consider your [users](#), [contributors](#), [reviewers](#), [content size](#), [deployed docs](#), and [translated docs](#).

- **Users.** The way that you organize your information must not detract from the user experience. Think about how users will read each page, navigate through the site, and find what they need. To make the content useful to users, you can interview users, complete card-sort tests, and use other methods to organize your site. For example, if you decided to create books, be sure that your users actually *need* books, and that you aren't just doing it because you are used to books and they are easier to maintain.
- **Contributors.** If developers are struggling to write docs in their own repo, consider a separate repo for docs. But, if technical accuracy is the higher priority, keep the docs in the dev repo and train developers to cultivate empathy for their end users who must understand the underlying layers of the software or service. For example, when only a handful of people know the best practices for a complex

product, get those contributors together to hash out the best content they can. You'll be surprised at how lively the discussion can be when the contributors' goal is to help others understand a complex system.

- **Reviewers.** When you create repos for docs, you must consider which collaborators are the best reviewers of the technical content. These collaborators are your main reviewers. Are these reviewers willing to set up content-change notifications in only a single repo or across multiple repos? If a cultural difference exists between docs reviews and code reviews, you must explain the docs review guidelines to your reviewers. You must also trust their judgment, especially if the continuous integration (CI) and continuous delivery (CD) processes publish the repo to the public web.
- **Content size.** How important is it to have a comprehensive and cohesive deliverable that the entire review team knows? If all reviewers must know the entire doc base, it might be useful to put all the docs in a single repo. However, if you need deep reviews on technical content that developers know best, you can house some deliverables in a separate docs repo and use web design to deliver seamless output.
- **Deployed docs.** If you must tightly coordinate and release the deliverables together, use a single repo with branches or tags to indicate a release point. If you need specialists to maintain some information, separate that

information into a separate docs repo and assign those specialists to that repo as its review team.

- **Translated docs.** If you must *freeze* the docs at a certain point to ensure that teams can translate without worrying about subsequent doc changes, you might split docs into a separate repo so that you can version the docs separately from the code.

Inclusion mechanisms

Although repo placement is an important part of information architecture decisions, also consider that some static site generators offer inclusion. Inclusion enables authors to insert a reference to other content and pull that content into the location at build time. This type of information architecture involves reuse of smaller pieces of information.

For API or reference documentation, you can write excellent examples with in-line reference information. In some cases, the reference information can be maintained as doc strings in the code itself.

The Django documentation provides an example of these reference pointers and inclusion methods. Django is a Python web framework. The Django docs are maintained in a `/docs` directory on GitHub. Contributors use Sphinx extensions to write RST examples for Django API docs.

For example, the following doc snippet uses the `.. currentmodule::` directive to point to reference documentation:

```
python
.. class:: QuerySet(model=None, query=None,
using=None)
```

Usually when you'll interact with a ``QuerySet`` you'll use it by :ref:`chaining filters <chaining-filters>`. To make this work, most ``QuerySet`` methods return new queriesets. These methods are covered in detail later in this section.

The ``QuerySet`` class has two public attributes you can use for introspection:

```
.. attribute:: ordered
```

``True`` if the ``QuerySet`` is ordered – i.e. has an :meth:`order_by()` clause or a default ordering on the model. ``False`` otherwise.

```
.. attribute:: db
```

The database that will be used if this query is executed now.

```
.. currentmodule:: django.db.models.query.QuerySet
```

Here is the [QuerySet API](#) web page:

QuerySet API

Here's the formal declaration of a **QuerySet**:

```
class QuerySet(model=None, query=None, using=None)[source]
```

Usually when you'll interact with a **QuerySet** you'll use it by [chaining filters](#). To make this work, most **QuerySet** methods return new querysets. These methods are covered in detail later in this section.

The **QuerySet** class has two public attributes you can use for introspection:

ordered

True if the **QuerySet** is ordered — i.e. has an `order_by()` clause or a default ordering on the model. **False** otherwise.

db

The database that will be used if this query is executed now.



Note

The **query** parameter to **QuerySet** exists so that specialized query subclasses such as **GeoQuerySet** can reconstruct internal query state. The value of the parameter is an opaque representation of that query state and is not part of a public API. To put it simply: if you need to ask, you don't need to use it.

Language: **en**

Methods that return new QuerySets

Documentation version: **1.10**

Django provides a range of **QuerySet** refinement methods that modify either the types or results

Use case: OpenStack docs

OpenStack docs were born in 2010 using [Bazaar](#), a version-control tool that is tightly coupled with Ubuntu's collaboration base. OpenStack moved from Bazaar to Git a few years later.

From the beginning, we knew we wanted to fit in with the OpenStack governance model and have a seat at the contributor table. The best way to get the most technical content was to work with the developers themselves, especially when there were only about two dozen of them. We had two documents, both API specifications written by architects.

Anne was the docs lead at the start of the project. She initially placed those XML-based documents in a single repo because the developers, who were completely immersed in Python, did not want to maintain XML. Because she expected to work with distributions, such as Red Hat that used DocBook at the time, she went ahead with the XML and DocBook plan and gathered contributors who could write really well in an XML workflow.

We automated the building and publishing with a Maven build system, and went on our merry way adding not just API docs but also installation and configuration information.

OpenStack grew by about one project a release for a while, adding up to about six projects in a couple of years. But fast forward a few more years and we now have nearly 30 services

and over 100 Git repos in the GitHub OpenStack namespace.

We have continued to keep a single `openstack-manuals` repo that builds install guides, configuration references, administration guides, user guides, and some specialty guides like operations, high-availability configuration, architecture and design, networking configuration, and command-line reference.

All those guides are built to docs.openstack.org. A few times we have split out a guide when a core group of reviewers assembled around the content. A few times we moved a guide back to the `openstack-manuals` repo when the core team could no longer maintain it.

A separate repo builds the developer.openstack.org website with an application developers guide. API docs have moved to the repos where docs specialists might not be able to merge docs. Mostly docs specialists can merge docs in the `openstack-manuals` repo.

We continue to work on the site architecture. One of Anne's biggest concerns lately is a *leaky abstraction*, where readers can see the implementation details and limits. This leak happens because the earlier OpenStack docs theme does not provide navigation for project documentation and the docs do not connect to the project's use cases.

How can the team ensure a great user experience while maintaining accuracy and speed? We keep looking for optimizations on both the source and output sides.

Summary

Information architecture and user experience go hand-in-hand. Treating docs like code should not limit your ability to provide well-organized docs. Take time for up-front planning and organization but also acknowledge that you might need to change repo locations, review teams, and deliverables over time.

Specialty information: REST API docs

The docs-like-code technique provides specific wins for REST API docs.

The growth of REST APIs drives workflows that make it easy for experts to collaborate to improve the usability of APIs.

Sometimes, an API's scale forces this collaboration. For example, when your small team of writers cannot produce docs for hundreds of API calls, you must garner the support of developers and developer experience (DX) experts with specialty knowledge.

GitHub workflows maximize the productivity of these experts to enhance your API docs by letting multiple contributors work on the same files at the same time. Plus, when you use standard doc tools to design an API, you make a more sensible, usable API. Tools like OpenAPI, formerly Swagger, help you manage more than just the docs life cycle for a usable REST API. GitHub and version control systems are a small part of the tools' ecosystem that provides excellent API and developer docs.

To determine how many resources and which tools that you need to document an API, evaluate its complexity, the best source framework, life cycle considerations, and which user docs you need.

- [API complexity](#)
- [API source frameworks](#)
- [API life cycle](#)
- [API user docs](#)
- [API guidelines](#)
- [Use case: OpenStack REST APIs](#)
- [Summary](#)

API complexity

To plan appropriate contributor resources for your API docs, you must analyze the complexity of your API. An API's complexity is not determined solely by how many methods it has. A simple REST API might have ten methods but each method might have dozens of parameters.

To evaluate an API's complexity, multiply how many methods it has by how many parameters, headers, and error codes each method has. Then, match the number of assigned development and docs resources to the API's complexity.

To complete that analysis, consider these REST API components:

Methods	<p>An API can provide HTTP GET, PUT, POST, PATCH, DELETE, and other methods.</p> <p>Some method implementations can be quite complex, such as a GET method for a resource collection that uses multiple query parameters for filtering and pagination.</p>
Headers	<p>In addition to standard HTTP request and response headers, look for any extra headers that the API defines.</p>
Parameters	<p>Your users must know which query, template, and body parameters are required, the default values for any optional parameters, and the constraints for parameters, such as whether a string value has a maximum number of characters.</p> <p>Additionally, users need to know how to use response fields from one API call to make subsequent API calls.</p>
Error codes	<p>Does the API use standard HTTP status codes?</p> <p>Are any associated error messages easy to understand? Or do you need to document additional information for each error code and message?</p>

API source frameworks

After you determine your API's complexity, choose a source framework for your API. An *API source framework* is also known as an *API description language*.

The source framework that you choose helps you design the API and enables you to generate both documentation and client SDKs in multiple languages.

To specify a large REST API, ensure that you can split the long specification into separate files for ease of maintenance. Also consider the editing environment. Try out the [online Swagger Editor](#), for example.

Following are the most highly adopted frameworks:

Framework	Description
OpenAPI (formerly known as Swagger)	RESTful APIs are represented as JSON objects that conform to the JSON standards. YAML, being a superset of JSON, can be used to represent a Swagger specification file.
RESTful API Modeling Language (RAML)	A YAML-based language that describes RESTful APIs.
API Blueprint	A documentation-oriented web API description language based on Markdown. In addition to the regular Markdown syntax, API Blueprint conforms to the GitHub Flavored Markdown syntax .

Why waddle when you can swagger?

A great blog post about the origins and context of the Swagger specification is by Tony Tam, former Reverb CEO: [Why WADL when you can swagger?](#)

API life cycle

When you evaluate which tools and resources you need to produce your REST API docs, you must determine your priorities for ensuring the API's success. Docs are not the *only* consideration but they are certainly an important success indicator for API use.

GitHub does not provide all the crucial life cycle systems that you need to deliver an API, though it can certainly provide data storage and retrieval, version control, and OAuth integration. Typically, an API management system provides security, parameter testing, monitoring, monetizing, visualizing, webhooks, terms of service, discovery, and the voice for the API itself. Examples include 3scale API Management Platform by Red Hat and Amazon API Gateway. SwaggerHub is an API development life cycle product with integrations. Integration starts with the OpenAPI standard itself, pointing to the importance placed on documentation for APIs.

To understand all aspects of an API life cycle, see [Subway Map API by API Evangelist Kin Lane](#).

API source frameworks fit into the larger picture of API life cycle management. It is an exciting time for REST API frameworks, and companies who solve this problem well are snapped up in acquisitions.

- In September 2015, SmartBear Software, a testing and development tool provider, acquired the Swagger API open source project from Reverb Technologies.
- In June 2016, Red Hat acquired 3scale, one of the early providers of API management solutions.
- In September 2016, Google acquired Apigee.
- In January 2017, Oracle acquired API management startup Apiary, which supported the API Blueprint format.

All this activity points to major business drivers racing to enable APIs managed as a strategic asset. Doc platforms are a crucial piece of that modern IT delivery and support. In this business context, owning the API documentation is crucial for standards-setting and the future influence of the direction of API management.

API user docs

API docs, which make the API findable and usable, are a leading indicator of an API's success. Great API docs offer a pre-sales opportunity and integration possibilities that you might not have considered. In [2013, the ProgrammableWeb ran a survey](#) that ranked “Complete and accurate docs” as the most important factor in making decisions about using an API.

While auto-generated reference information can provide a starting point for your users, these docs do not provide enough information to on-board users and provide ongoing support. Users must understand the API as a product that helps them accomplish a task.

In addition to auto-generated reference docs, give your users a getting started guide with use cases, examples, and authorization and authentication details. You need docs that describe the architecture and why the API exists and that provide code samples and tutorials. Some great examples are the [Stripe REST API docs](#) and the [Parse REST API docs](#).

API guidelines

When you use GitHub for REST API docs, it's important to document your expectations for source file markup, contribution rules when documenting with code itself, completeness definitions, review guidelines, and quality indicators.

In OpenStack, the [docs contribution guidelines](#) and the [API design guidelines](#) produced by a collaborative API working group include the REST API docs guidelines.

Having the guidelines in two places with pointers to each gets the attention of both specialty writers and developers of the APIs, as well as consumers of the APIs.

Use case: OpenStack REST APIs

The Web Application Description Language (WADL) is an XML-based standard submitted to the World Wide Web Consortium (W3C) in 2009. In 2010, OpenStack opted to use the WADL format to document the OpenStack-defined Object Storage and Compute REST services.

XML tools specialists at Rackspace ensured that OpenStack contributors could validate developers' requests for the cloud service and keep the docs true to the implementation. The format and tooling provided a win-win situation.

In 2014, the WADL specification had not been adopted as a standard, the Rackspace doc-specific tools team was disbanded, and the ability of a single toolset and WADL-based format to filter requests and document the services proved unwieldy. Plus, the community was unable to make required doc updates by using the unnecessarily complex XML- and WADL-based toolset.

The OpenStack team recognized that it needed a new API reference doc solution. For a full year, Anne worked with others on ways to transform the API documentation. These were the main problems:

- Historically, API reference information has been difficult for community and company resources to maintain due to the specialized nature of the toolset and the REST API.
- API reference information must live where API writers, API developers, contributor developers, and the API working group members can review it together.

To solve those problems, developers tried to migrate the dozen or so WADL files to OpenAPI (Swagger). The partial migration left the references incomplete. People also tried to use OpenAPI to manually write the rest of the specification. For some APIs, this tactic could work, but not for all OpenStack APIs.

After the team looked at OpenAPI as a way to document a large API like Compute (over 900 methods), it decided that OpenAPI could not be retrofitted for an already-existing API. Instead, the team opted to use reStructuredText (RST), Yet Another Markup Language (YAML), the Sphinx build framework, and extensions ([os-api-ref](#)) and web development in a Sphinx template ([openstackdocstheme](#)) to get decent display and interaction with the reference parts. And, by using these standard formats that OpenStack developers already used heavily, the team made it easy for developers to contribute to the REST API docs. As an example, look at the [RST source](#) for a single REST API method:

```
List Servers
```

```
=====
```

```
.. rest_method:: GET /servers
```

```
Lists IDs, names, and links for all servers.
```

The `.. rest_method::` points to a Sphinx extension that creates the HTML and interaction for the web page.

Here is an example snippet of the RST that points to YAML to document the request parameters:

```
Request
```

```
-----
```

```
.. rest_parameters:: parameters.yaml
```

- limit: limit
- marker: marker
- sort_key: sort_key_server
- sort_dir: sort_dir_server

The [parameters.yaml](#) file describes the [sort_key_server](#). The output is in a nice, readable HTML table, and because `required` is marked `false`, the output indicates that it is an optional parameter.


```

sort_key_server:
  description: |
    Sorts by a server attribute. Default attribute
    is ``created``. You can specify multiple pairs of
    sort key and sort direction query parameters. If
    you omit the sort direction in a pair, the API uses
    the natural sorting direction of the server
    ``sort_key`` attribute.
  in: query
  required: false
  type: string

```

In 2017, as Anne looks back at the two-year REST API migration effort, she still has confidence in the final solution for this difficult area of documentation. The large [Compute API](#) site offers great examples of the new API reference, maintained by the project team rather than a centralized and specialized docs team.

developer.openstack.org/api-ref/compute/?expanded=list-servers-detail

GET /servers
List Servers

Lists IDs, names, and links for all servers.

Servers contain a status attribute that indicates the current server state. You can filter on the server status when you complete a list servers request. The server status is returned in the response body. The possible server status values are:

- ACTIVE**. The server is active.
- BUILDING**. The server has not finished the original build process.
- DELETED**. The server is permanently deleted.
- ERROR**. The server is in error.
- HARD_REBOOT**. The server is hard rebooting. This is equivalent to pulling the power plug on a physical server, plugging it back in, and rebooting it.
- MIGRATING**. The server is being migrated to a new host.
- PASSWORD**. The password is being reset on the server.
- PAUSED**. In a paused state, the state of the server is stored in RAM. A paused server continues to run in frozen state.
- REBOOT**. The server is in a soft reboot state. A reboot command was passed to the operating system.
- REBUILD**. The server is currently being rebuilt from an image.
- RESCUED**. The server is in rescue mode. A rescue image is running with the original server image attached.
- RESIZED**. Server is performing the differential copy of data that changed during its initial copy. Server is down for this stage.
- REVERT_RESIZE**. The resize or migration of a server failed for some reason. The destination server is being cleaned up and the original source server is restarting.
- SOFT_DELETED**. The server is marked as deleted but the disk images are still available to restore.
- STOPPED**. The server is powered off and the disk image still persists.
- SUSPENDED**. The server is suspended, either by request or necessity. This status appears for only the XenServer/XCP, KVM, and ESXi hypervisors. Administrative users can suspend an instance if it is infrequently used or to perform system maintenance. When you suspend an instance, its VM state is stored on disk, all memory is written to disk, and the virtual machine is stopped. Suspending an instance is similar to placing a device in hibernation; memory and vCPUs become available to create other instances.
- UNKNOWN**. The state of the server is unknown. Contact your cloud provider.
- VERIFY_RESIZE**. System is awaiting confirmation that the server is operational after a move or resize.

Normal response codes: 200
Error response codes: badRequest(400), unauthorized(401), forbidden(403)

Request

Name	In	Type	Description
limit (Optional)	query	integer	Requests a page size of items. Returns a number of items up to a limit value. Use the limit parameter to make an initial limited request and use the ID of the last-seen item from the response as the marker parameter value in a subsequent limited request.
marker (Optional)	query	string	The ID of the last-seen item. Use the limit parameter to make an initial limited request and use the ID of the last-seen item from the response as the marker parameter value in a subsequent limited request.
sort_key (Optional)	query	string	Sorts by a server attribute. Default attribute is created . You can specify multiple pairs of sort key and sort direction query parameters. If you omit the sort direction in a pair, the API uses the natural sorting direction of the server sort_key attribute.
sort_dir (Optional)	query	string	Sort direction. A valid value is asc (ascending) or desc (descending). Default is desc . You can specify multiple pairs of sort key and sort direction query parameters. If you omit the sort direction in a pair, the API uses the natural sorting direction of the direction of the server sort_key attribute.

By using a format and toolset that the community can maintain and that can produce better web experiences, the OpenStack team can offer the most accurate docs across more REST API services.

Summary

Specialty docs like REST API docs require careful analysis and planning. Simply moving to a docs-like-code model does not guarantee the success of your API docs. The right approach is to design your API, plan your docs, and then integrate standards and guidelines into your docs.

What about wikis?

Although wikis don't meet all the criteria of a docs-like-code framework, they do provide a simple and collaborative authoring environment with a low barrier to entry. That said, GitHub and other collaborative systems built on `git` provide project wikis. Often projects use those wikis in a docs-like-code way, where files are checked in to publish them to the wiki.

Using a project wiki within a Git repo may make sense for your project, whether hosted on GitHub, BitBucket, or GitLab. Nick Volynkin, a technical writer at Plesk, makes these excellent points while advocating for project wiki use in a docs-like-code environment.

- Project wikis offer an easy onboard for contributors, and are a viable choice for relatively small projects. However, as the project evolves, you will likely need a full-fledged documentation website.
- Wikis can be useful for rare use cases or edge test cases, and provide troubleshooting tips for those cases. In the project wiki, you can place the topics that are not relevant for most readers or have no place in the main documentation.
- Wikis can also host working drafts, team jokes, project folklore, or simple feedback pages. These pages wouldn't have to meet the standards and style guide of the main

documentation, but can certainly add to the project's sense of identity.

So, why not use a wiki instead of GitHub as an authoring and content management system? This section compares the systems in some key areas.

Usability

Wiki	Many developers never leave their Terminal window and do not want to open a browser window to log in and edit a wiki page that they must find in the first place.
GitHub	You can avoid the context switch from coding to opening a wiki page by placing the docs directly in the code or in the same GitHub repo as the code.

Security

Wiki	Sometimes wikis are seen as ideal internal-only documentation sites because the wiki is accessible behind a firewall, for example.
GitHub	GitHub Pages from GitHub.com are publicly available when published, even when published from a private repo. You can work around this issue by implementing an authentication workflow on a site that you upload to GitHub Pages, but you must have the web development resources to maintain the login requirement. You could use GitHub Enterprise Pages, which requires a VPN connection to view pages.

Statistics

Wiki	Some wiki engines give you statistics that help you determine who is a topic expert. Others may only provide statistics to users with certain permissions.
GitHub	GitHub enables you to see which contributors are working on a particular part of the software project, which can help with documentation needs.

Automated builds

Wiki	Many wikis only provide a simple Save button per page. With additional plugins for Confluence, for example, you can automate parts of the publishing process and put a workflow for reviews in place.
GitHub	Treating docs like code lets you automate more tasks than simply rendering and publishing HTML. Automated builds for review purposes are a big advantage of treating docs like code.

Review workflow

Wiki	Wikis often have add-ons for review workflows, but when the heart of a wiki is quick editing, people do not switch easily to the review workflow.
GitHub	In GitHub, it's expected to let others review your changes before merging, or publishing in the case of a document.

Web design

Wiki	If your web development resources know a particular wiki framework really well, they might be able to create a nicer end-user experience for a wiki than in a highly flexible web framework like Sphinx or Jekyll.
GitHub	When publishing an entire site using static site generators, you get more flexibility in choosing web designs and navigation than with most heavily opinionated wiki frameworks.

Reuse

Wiki	Wikis are page-oriented, and to do releases, you might need to publish a page twice. Over time in a wiki, contributors create new pages instead of looking for and editing existing pages, and trusted content becomes harder to find. Sprawl can be a problem with both solutions, but the organized nature of having documentation near code seems to keep the two in lock-step when they share systems such as version control and review workflows.
GitHub	With GitHub, you can back port a change from one release to another by using the same pull request workflow.

OPTIMIZE DOCS-LIKE-CODE WORKFLOWS

Git has a number of workflows to match the way that your team wants to work, whether you use GitHub, GitLab, Bitbucket, or another Git-based system. If your team either wants to start treating or is already treating docs like code, the workflow choice might already be made.

The following basic workflows in GitHub, for example, help you collaborate with others in your repo. They are ordered from simplest to most complex.

- **Centralized.** Work in a central branch, typically named `master`. Always push changes to that branch when working with others.
- **Feature branch.** All feature development takes place in a dedicated branch instead of the `master` branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main code base. It also means that the `master` branch never contains broken code, which is an advantage for continuous integration environments.
- **Gitflow.** Work in feature branches, but have agreed-upon guidelines for branch interaction.

- **Forking.** Work independently in a fork of the repo. Merge as often as you like to your fork. Request merges through pull requests to the centralized repo (`master`) to integrate features.

Note: The Gitflow workflow uses feature branches, but you can also combine a centralized or forking workflow with feature branches. You do your work locally in a dedicated branch instead of working in the `master` branch. Then, when you create a pull request, you can request to merge your feature branch to the upstream `master` or another branch, depending on your workflow.

For more information, see the [Comparing Workflows](#) tutorial from Atlassian.

If the docs are in the code repo and that repo has a workflow, use the existing code workflow. If your docs are in a separate repo with a different team of reviewers, work with your team to choose a workflow.

Realize that the further the docs are from the code, the more difficult it can be to keep in sync. Think about creating a culture where code changes don't merge without corresponding doc changes. For example, use the settings in a GitHub repo that enable doc builds from a `/docs` directory on the `master` branch, which would then allow teams to review and require docs changes with any code changes before merging.

Your team's history and culture can partially dictate workflow choices. For example, does your team use a pessimistic-merge version control system, where the system locks entire files? Or, is the team well-tuned to the optimistic-merge view and using Git natively, where merges are considered fairly easy to resolve? When you write docs like code with a new team and new repo, you can start with a simple workflow and move to a more complex one.

Does the team have a great review culture, where the members help each other and even patch each other's patches? Or, is the team more proprietary, where each member owns a set of content that no one else touches? With a team that prefers ownership, you can use *feature branches*.

Technology constraints also dictate the workflow. Is there a technical reason for release code to be protected from the public for a while? You might need to have people working independently for a while, so *forking* makes sense. Is the centralized code base complex? Is it in a single repo or does it split across many repos? With a central branch, you might want to publish early and often, so protective measures aren't necessary.

The hardest part about choosing or switching workflows is knowing when you can, or can't, merge to `master`. It's easy to develop *muscle memory* and type `git merge origin master` even when you do not want to do that!

Use the following decision table along with the information that follows the table to decide which workflow might work best for your team.

	Centralized	Feature branch	Gitflow
Are the docs in their own repo?			
Docs in separate repo	✓	✓	✓
Docs in code repo	Use dev workflow		
How will you publish?			
CI publishing	✓	✓	✓
gh-pages publishing	✓		
Will contributors preview docs locally or on a server?			
Preview docs locally	✓		
Preview docs on server	✓		
How will you collaborate with others?			
Low collaboration	✓		
High collaboration		✓	✓
How many pull requests and reviews do you expect?			
Small number of PRs	✓		
Large number of PRs		✓	✓
How often do you release your docs?			

Docs release with code	Use dev workflow		
Docs release continuously	✓	✓	✓

Are the docs in their own repo?

If docs are in their own repo, you can version and release them separately from the code. You can choose a centralized, feature branch, or forking workflow.

If your docs team can choose its workflow, consider keeping it simple and using a centralized workflow. With docs in their own repo, you can set up access control so that the review system allows only the lead developers or lead writers to publish, for example.

Also, if you are using GitHub, consider using only the `master` branch only for both editing and publishing. This tactic works well if you are constantly publishing and want the `master` branch to consistently reflect the current state of the docs.

GitHub has three settings for publishing docs, so read [Configuring a publishing source GitHub Pages](#) for details as you research workflows.

When docs are integral to the code, the docs workflow must match the development workflow because the collaborators are using the workflow that is best for that repo. You can still advocate for stable branches in the code repo for docs only, and with branch publishing, you can publish only the docs directory.

Beyond just the workflow, you must also consider the information architecture choices that individual repos give you. If the repo makes a single website, your architecture

depends on the way the content is rendered. If multiple repos create content for a single website, you must have a web design that enables readers to view output from multiple repos.

How will you publish?

Without any sort of outside continuous integration (CI) tooling, you can use the `gh-pages` branch that GitHub provides for actual publishing. Using the `gh-pages` branch means that your workflow needs to include merging to `gh-pages` at specific times for publishing purposes.

In repos that represent online docs, you can make the `master` and `gh-pages` branches identical with a simple configuration setting. Then, you treat `gh-pages` as if it were the `master` branch, and you continuously publish with a simple centralized workflow. You can change which branch is used for publishing with the Settings tab for the repo.

You can publish in this way only if you are using Jekyll plug-ins that support `gh-pages`. If you have plug-ins that `gh-pages` can't publish, you must build on `master` and copy the files from `_site` as the root of the `gh-pages` branch. That said, you are required to build from the `master` branch when building a user or organization site. For details, see [GitHub Pages docs](#).

[Octopress](#), a fork of Jekyll, has a nice command-line tool to manage pushing to the `gh-pages` branch.

Will contributors preview docs locally or on a server?

With either a centralized or forking workflow, you should determine how contributors can view rendered docs while working on patches. Setting up a staging server could be time-consuming and require development resources. Can contributors easily stage changes, such as by using GitHub Pages? For example, could contributors set up their forks to publish to `http://username.github.io/repo-name/` to preview their changes, without affecting SEO or the domain name for the main site? If your docs repo is a private repo, are you willing to let the staged pull requests be found on the Internet prior to actual publishing? If not, then you likely don't want to use the forked `gh-pages` method for a staging server for your docs site.

Instead, provide contributors with instructions for building locally so that they can preview their changes before submitting a pull request. For example, the Apache jclouds docs build each pull request to a Rackspace Cloud Files staging location for a preview of the patch changes. A message from the build job tells the [contributor how to preview the doc change](#).

With a centralized workflow, contributors need to see a preview before pushing to `master`. With a forking workflow, ensure that the preview meets the privacy needs of the source

docs.

How will you collaborate with others?

Even with a large team, you can give contributors access to the publishing branch as long as your contribution guidelines are very clear. If you don't expect many reviews, you can pick a simpler workflow. If you expect many contributors to add features to the docs, you might choose the feature branch workflow.

How many pull requests and reviews do you expect?

Whether you expect to review 20 pull requests daily (that's a lot!) or one each week or month, you might want to track what is merged. If you have a small team with only a few reviews a week, it's fairly easy to remember which topics have changed or how the overall site itself has changed in a given time period. In this scenario, your workflow choice can be a simpler one.

If you have a lot of contributors and want to keep the published site fairly up-to-date, you likely want a staging environment and a production environment with branches for each. In this scenario, the feature branch workflow or the Gitflow workflow would work well.

How often do you release your docs?

If your docs must be released only at the same moment as the product is available, you likely want to merge to a separate branch for the duration of the work leading up to the release. With this consideration, a forking workflow works best for release docs.

If you must publish the site as soon as a change lands, you can use a simple workflow and even delete the `master` branch and use just the `gh-pages` branch.

You can also take a hybrid approach as your docs set grows. For example, OpenStack has release deliverables written for a single release every six months, and continuously publishes deliverables written to span multiple OpenStack versions. Release-specific books, such as the installation guides and configuration guide, reside on two currently supported branches (the current release and two previous releases). Ongoing writing work continues on the `master` branch. The continuously released books, such as a user guide and administration guide, are built only from the `master` branch.

Summary

The workflow that you choose depends on the history of the team and repos where they have been working. At first, a simple workflow, such as the centralized workflow, can work well. As the number of contributors, reviewers, releases, and publishing locations grows, you can add complex feature branch or forking workflows.

Author and build content

We are here to write, after all. We author content to convey concepts, teach others, document how to do an important task, or describe required reference information.

- [Choose an editor](#)
- [Build locally first](#)
- [Choose a static site generator](#)
- [Example source file and outputs](#)
- [Summary](#)

Choose an editor

To author the source files that you store in GitHub or another code system, use a lightweight markup language such as [AsciiDoc](#), [Markdown](#), or [reStructuredText](#) (RST).

To enable many collaborators, choose a platform-independent, text-based editor. You can invoke many text editors from the command line in a Terminal window or in a web browser.

Authors who are accustomed to an integrated development environment (IDE) might find that they can configure plug-ins for docs markup languages. Others who are constantly in their Terminal window and don't want to context-switch to a separate editor find `emacs` or `vi` to their liking and work on docs without losing context or switching tools.

You can use side-by-side editors to author simple ASCII markup and get an approximation of the output. Typically, you must be a bit skeptical of the accuracy of that output, especially for Markdown because it has a few interpretations. Still, many writers prefer the side-by-side editing that tools like [Mou](#) and [Macdown](#) for Markdown and the [Online reStructuredText editor](#) for RST offer. The online RST editor does not show all extended semantic markup exactly how your CSS output does, but it's a good working environment for many people. This feature is especially helpful when you are editing tables in simple markup, which is a challenge.

Markdown controversy resolved

In March 2017, GitHub published a formal spec for GitHub-Flavored Markdown. This publication offers a resolution to the difficulty in using Markdown for technical documentation that Eric Holscher, founder of Read the Docs discusses in a [blog post a year earlier](#).

From the [GitHub blog post](#): “This formal specification is based on CommonMark, an ambitious project to formally specify the Markdown syntax used by many websites on the internet in a way that reflects its real world usage. CommonMark allows people to continue using Markdown the same way they always have, while offering developers a comprehensive specification and reference implementations to interoperate and display Markdown in a consistent way between platforms.”

Now you can analyze whether a tool or parser adheres to a particular spec.

Build locally first

When you author content, you want to see the output just as your end users will see it. As mentioned in the preceding section, some side-by-side editors let you view the output as it appears in a web browser, but that output is not always reliable.

Ideally you should have a staging environment for the docs. When you first start treating docs like code, though, you might not have the tools in place for a staging environment. If you don't have a staging environment, you must be able to build docs locally. With Ruby-based and Python-based environments, you need a development environment.

On Mac OS X, use [Homebrew](#) to manage packages and dependencies. Homebrew also has nice features like the `brew doctor` command to help troubleshoot problems with permissions or symbolic links. Using `brew` ensures that your system Python, for example, remains untouched, while you can run different versions of Python.

openstack-doc-tools

OpenStack maintains a separate repository, [openstack-doc-tools](#), that provides the scripts that build the docs.

Those tools are used by the infrastructure environment and can also be used locally on a Mac or Linux-based system that has Python installed.

Choose a static site generator

Static site generators work great to generate web and other outputs from source files that you store in GitHub and author in lightweight markup languages like Markdown or RST. As of December 2016, you can choose among nearly 450 static site generators. For comparisons, see [Static Site Generators](#).

What if you never build the docs?

GitHub provides an HTML rendering of many ASCII markup pages with [GitHub Flavored Markdown as the supported syntax](#). It's unclear whether web analytics for rendered doc pages on GitHub are available. Many projects don't even bother to render their docs on a separate website. There is no navigation for reading source docs on GitHub. Although it's possible to only write docs and never publish them, you should consider your audience's needs before deciding to do so.

Example source file and outputs

These examples show an RST source file and two types of output. Python developers use RST to document their code inline. They use RST-based rendering engines, such as Sphinx, to render web content.

RST source file

This example shows an RST source file. When reviewing simple markup, you can easily see what has changed. Also, the whitespace in RST is as meaningful as the text itself.

```
=====
Discover the version number for a client
=====
```

Run the following command to discover the version number for a client:

```
.. code-block:: console
```

```
$ PROJECT --version
```

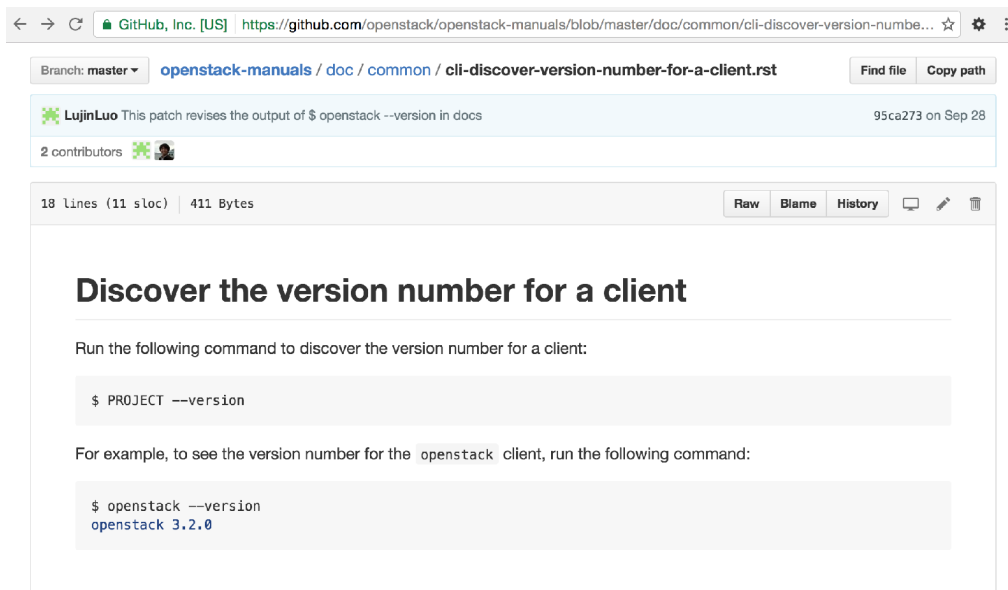
For example, to see the version number for the ``openstack`` client, run the following command:

```
.. code-block:: console
```

```
$ openstack --version
openstack 3.2.0
```



GitHub-rendered output


You can view an RST source file as rendered content on GitHub, as shown in this example. Markdown is also prevalent when working on GitHub, and might render better than RST in some cases.



The screenshot shows a GitHub web interface for a file named `cli-discover-version-number-for-a-client.rst` in the `openstack-manuals` repository. The file is on the `master` branch. A patch by `LujinLuo` is visible, with the description "This patch revises the output of `$ openstack --version` in docs" and a commit hash of `95ca273` on Sep 28. The file statistics show 18 lines (11 sloc) and 411 Bytes. The rendered content features a heading "Discover the version number for a client", followed by instructions to run a command to discover the version number. Two code blocks are shown: one with the command `$ PROJECT --version` and another with the command `$ openstack --version` followed by the output `openstack 3.2.0`. The interface includes navigation tabs for "Raw", "Blame", and "History", as well as buttons for "Find file" and "Copy path".

Branch: `master` `openstack-manuals` / `doc` / `common` / `cli-discover-version-number-for-a-client.rst` Find file Copy path

 LujinLuo This patch revises the output of `$ openstack --version` in docs 95ca273 on Sep 28

2 contributors 

18 lines (11 sloc) 411 Bytes Raw Blame History

Discover the version number for a client

Run the following command to discover the version number for a client:

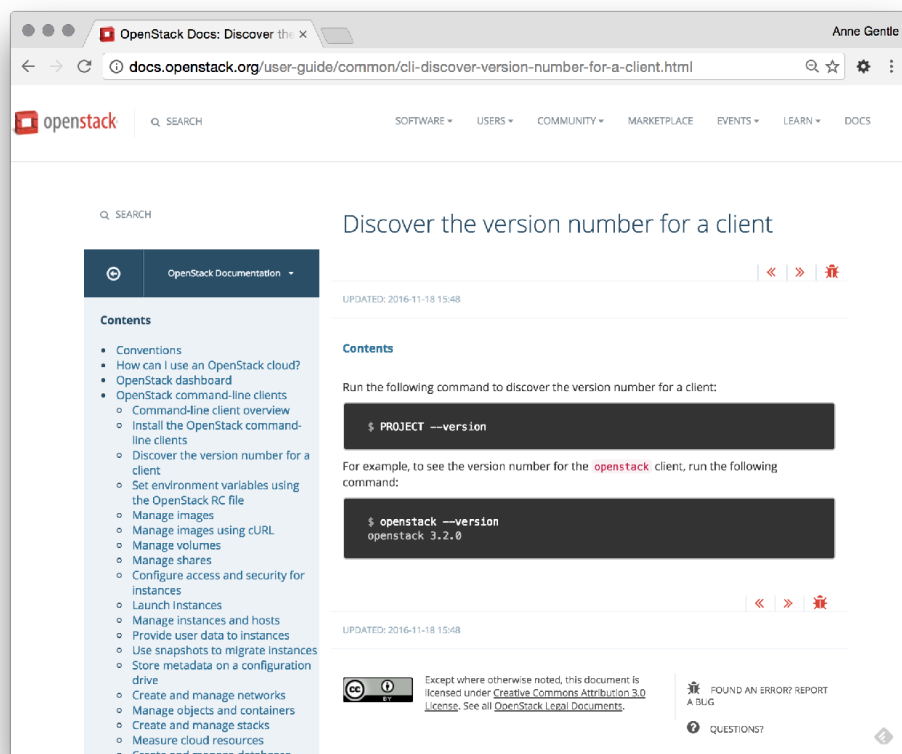
```
$ PROJECT --version
```

For example, to see the version number for the `openstack` client, run the following command:

```
$ openstack --version
openstack 3.2.0
```

Website-rendered output

In addition, you can view the rendered content on a website, such as the OpenStack site, as shown in this example. This website rendering is done with a Sphinx theme.



Summary

When you treat docs like code, your contributors must be able to author content easily and efficiently. To encourage more contributions, provide an easy-to-use authoring tool with a side-by-side comparison feature, coach great writing, enable both local and staged builds, and make sure that the deployed docs are excellent.

Automate builds

End the tedium, enable the robots.

Recite this mantra when you choose CI/CD tools.

- Continuous integration (CI) means that code is continuously tested, integrated with other code changes, and merged.
- Continuous deployment (CD) means that code is continuously deployed with each patch to the entire code base.

For docs, CI/CD means that content is continuously tested, merged with each patch, and deployed. Deploying, in this context, means publishing; for example, output files are copied to a web server for all to see.

OpenStack can merge over 900 docs changes in fewer than three months because they treat docs like code and continuously build and deploy reviewed content from multiple Git repos. If you already automate code builds and deployment, you can automate your docs builds and deployment.

- [Build and deployment tools](#)
- [CI/CD for docs](#)
- [GitHub Pages](#)
- [Programming language considerations](#)

- Additional uses of CI
- Summary

Build and deployment tools

When you have simple markup and flat files for your docs source, you need to build those files into HTML that's connected to CSS and JavaScript, so that you can deploy the files to the web server. When treating docs like code, *build* means to parse and render to another format, and *deploy* means copy the files to the correct location on a configured server. When making an EPUB, for example, you build HTML and compress the files with some additional descriptive files into a single artifact. The build steps can differ depending on the deliverable. The deploy steps can change based on the method of delivery. For example, you could build translation artifacts and then deploy those to a translation server.

To build this book, we used these tools:

Tool	Used to
TextWrangler, vi, and Atom	Edit Markdown files.
GitHub and Git	Store files and perform version control.
A GitBook webhook	Automate the draft website, PDF, and EPUB file outputs.

To make the docslikecode.com site, Anne used these tools:

Tool	Used to
TextWrangler and vi	Edit Markdown files.
GitHub	Store files and perform version control.
So Simple	Provide a nice, responsive web theme.
Jekyll	Create HTML.
GitHub Pages	<p>Automate builds from a <code>gh-pages</code> branch. The settings for automatic publishing for GitHub Pages is on the repo's Settings page.</p> <p>You can use the <code>gh-pages</code> branch, the <code>master</code> branch, or a <code>/docs</code> directory on the <code>master</code> branch for hosted GitHub Pages.</p>

CI/CD for docs

A docs build is typically triggered when updates are merged into the source code in the correct branch of a repo. A build usually consists of a bash script at the basic level, run by a deployment system such as Jenkins, Travis, or Circle CI. GitHub has many CI systems to choose from. You can learn more about them on the GitHub website at [Integrations/Continuous integration](#).

Benefits of using CI/CD for docs

OpenStack has multiple projects merging multiple changes every day, so the docs system needs to keep up with that many changes. CI/CD enables that, so it's not only a benefit but a requirement in that environment.

CI/CD also means that reviewers don't have to build docs in their local environments. When the system builds a review draft, casual contributors and reviewers can avoid the overhead of downloading the patch, replicating the build environment, and then building the docs. Instead, they can quickly see how a change is rendered when it is published. Because of the automation in the system, reviewers can review both the source and the output.

The speed of the builds increases because writers can work on multiple patches at once while the cloud-based CI/CD continues to run.

In OpenStack, using the same workflow that developer and infrastructure teams use makes it easy for developers to contribute to docs. The recent move to RST as the authoring format makes it even easier because RST is the common markup language used in OpenStack.

Avoiding automation risks and pitfalls

Considering that writing is both a technical venture and an artistic endeavor, you have to strike a balance between what should be automated and what requires a master crafter. Some risks of not striking this balance could be publishing too soon or publishing incomplete docs.

To mitigate these risks in OpenStack, the teams build trust among reviewers and have great in-person discussions about reviews at the six-month OpenStack summits. The docs team wrote a review guide and trains reviewers about using good judgment when reviewing patches. As long as reviewers have guidelines such as “It’s better than what we have now,” or “I’ve tested this and it works,” or “This doc fix matches the reported bug I’ve investigated,” the risks associated with publishing doc updates 50 to 100 times a day goes down. Automating the build and deployment processes gives reviewers confidence that the docs always build correctly and that the entire docs site is stable.

Some OpenStack manuals are release-independent, and some document the current release. For manuals that document a specific release, like the installation guides, the docs team

updates the files for each new release and works in branches. The released document is updated with critical changes, and the document for the next release gets the majority of changes and gets published in a hidden place for easy review of the current draft. A docs team core reviewer publishes the release-specific guides publicly after the software is released, and then contributors start working on updates for the next release.

GitHub Pages

GitHub offers a directly hosted solution for documentation called [GitHub Pages](#). This solution gives you a simple but powerful way to edit, push, and view live web pages.

You can register a domain name, create a repo, write a handful of pages, and then use GitHub Pages to publish to the domain name by simply pushing to a branch named `gh-pages`.

Because you can map a domain name to the entire site, or simply use the provided `{githubusername}.github.io` subdomains, you get nice web experiences branded for your project, organization, or company. The tutorial on the [GitHub Pages](#) site offers a quick way to try it out, and the tutorial in this book shows how to use a specific Jekyll theme to build a website by using GitHub Pages.

The basic premise is that when you push changes to the `gh-pages` branch, you initiate a build that becomes web pages. You can even delete the `master` branch for a docs-only repo and always edit and push to the `gh-pages` branch. If you use Jekyll with GitHub Pages, only certain gems are supported to ensure security of the hosted sites. However, you can copy the output HTML files and related CSS and JavaScript files to the root of the repo and push those files to the `gh-pages` branch. That method uses GitHub Pages to host a static site, which lets you use any gems or theme that you want.

GitHub Pages provides the following features:

- Automatically create site maps and an Atom feed
- Optimize the site for search engines, social media sharing, and redirects
- Ensure that common repo metadata is available to the site
- Enable the use of `@Mentions` of GitHub user names on sites built with GitHub Pages

This tool is a great enabler of fast publishing from any repo. GitHub Pages has created some nice efficiencies for hosting documentation from GitHub. Here are some additional tips:

- Be sure to note the specifics for repo or organization names when using a custom domain name, which is explained clearly in the GitHub docs on custom domains.
- Use the Settings tab for the repo to indicate whether to publish the `master` branch or `gh-pages` branch, or use a `/docs` directory on the `master` branch for GitHub Pages.
- If you use a repo only for a website, you can delete the `master` branch and maintain only the `gh-pages` branch, as long as your site always uses only the Jekyll plug-ins that the GitHub Pages gem supports. This simplification makes the purpose of the repo clear.
- When using a custom domain, you also create a `CNAME` file that indicates the domain name to GitHub Pages. Domain name registration costs about \$10 per year, and you must see your domain registrar documentation to set up the domain name. If your DNS provider does not support `ALIAS` records on the root apex (`@`), you must

create A records that point to 192.30.252.153 and 192.30.252.154 , which are IP addresses provided by GitHub.

GitHub Pages limitation

One current limitation of GitHub Pages is that you can't serve custom domain pages over HTTPS.

Programming language considerations

The two languages and docs frameworks most often used to build docs like code are Python with Sphinx, and Ruby with Jekyll.

When you learn Python, you realize how much readability, clarity, concise style, formatting, and opinion matter in this language. Whitespace is actually required and meaningful, unlike in other programming languages. Python is a dynamic programming language, and because it's interpreted, it often takes fewer lines of code to accomplish a task than in other, statically typed languages, such as Java and C++.

Ruby provides multiple ways to perform a task, unlike Python with its *one way is the most understandable way* philosophy. Ruby is dynamic, reflective, and quite general-purpose, and it predates Python by about four years.

You can do just about anything with either language. Both have excellent beginning frameworks for docs and websites. Both have a huge set of libraries for reuse. Both have a specialized tool chain to make lovely websites, and both offer strong community support and maintenance systems.

The maturity and wide adoption of both languages gives writers a choice. To choose, consider the developer expertise that you already have in case you must extend the framework,

such as by writing Sphinx extensions or Jekyll gems.

You might want to use a specific theme or organizational structure as you examine the web output. For the source, consider whether semantic markup is meaningful to your contributors and in your output. For semantic markup, reStructuredText (Python and Sphinx) has better support than Markdown (Ruby and Jekyll). Markdown has a problematic history with multiple interpretations for output, but now has a new [specification for GitHub Flavored Markdown](#) (GFM).

Python: RST and Sphinx

The source for docs built with Sphinx in the Python community is called [reStructured Text](#) (RST). It has an advantage over other simple markup languages because it provides semantic (meaningful) names to indicate whether inline markup is for a command or parameter element, for example. For an example of style guidance for RST markup, see the [OpenStack Contributor Guide's inline elements reference](#).

One amazing aspect of the Sphinx and Python community is the [Read the Docs](#) site, a community-maintained CI/CD system for documentation that supports both RST and Markdown as source. Started by Eric Holscher and other writers, the site hosts documentation for multiple projects, providing searchable and findable documentation pages. In addition to the community-supported site, Write the Docs conferences take place in North America and Europe each

year, and regional meet-up groups talk about great techniques for documentation. In fact, many presentations describe integrating documentation practices with code practices:

- 2017 US conference: [Jodie Putrino](#) presented “Treating documentation like code: a practical account” to share her experiences at F5 Networks.
- 2016 US conference: [A panel](#) of folks from Rackspace, Microsoft, Balsamiq, and Twitter talked about how they are adopting these practices.
- 2016 EU conference: [Margaret Eker and Jennifer Roundeau](#), and [Rachel Whitten](#) talked about docs-as-code in practice.
- 2015 US conference: [Riona MacNamara](#) spoke about how adopting docs like code has completely transformed how Google does documentation.

Ruby: Markdown and Jekyll

Source files for docs in the Ruby ecosystem are Markdown. GitHub has its own interpretation of Markdown, so any document files written in Markdown are easy to read on GitHub itself.

What if my source for documentation isn't Markdown or RST?

If your content can be served from a web server or uploaded as PDF or eBook-formatted files, you can deploy in an automated fashion and use a docs-like-code system for documentation authoring and deploying.

For example, OpenStack deployed HTML and PDF from DocBook source for years. Also, REST API documentation was created from WADL, an XML-based source file. The build simply built from XML source and copied the resulting HTML and PDF files through FTP.

Or, if your favorite markup source is AsciiDoc, there's a great write-up about [how to use GitHub Pages for Pelican](#) with tips for helper tools to update the `gh-pages` branch continuously.

Web development tools

To do the web development necessary to make great interactions on websites, you must layer in more languages and markup with these interpreted languages. The list includes:

- JavaScript
- CSS
- Less or Sass, which compiles the CSS using variables
- Node.js
- Dependency organizers and library installers

You don't need to have a complete understanding of every framework, but be sure you have a decent theme that provides a nice user experience.

Also, the [Pandoc](#) command-line tool is useful when you must convert from one source file to another, especially in bulk or when you must automate.

Container images for docs

Automated builds might be the best part of the docs-like-code approach. Automation means that your memory-intensive builds won't cause your old laptop fan to spin out of control or your computer to die a swift memory-error death.

In the beginning of building docs at Rackspace, the team had a system of button clicks for publishing, with a server built with the Apache jclouds Java SDK, and running a Maven plug-in. When Rackspace teams enabled more contributors, the doc-build automation system changed from a home-grown and quirky publishing system to containers.

Although containers are not new, they are recently improved for use by developers. Containers let you serve an already installed and running software stack, such as Ruby with Jekyll.

You can use a Docker™ image for docs to build a Jekyll site locally. With Docker, you launch a local virtual machine (VM) that runs a container already created with Ruby and Jekyll to build the site in the VM. The process is straightforward: install Docker, get a prebuilt container image, run a command, and

test your site through a URL that is served from a local container. Find an example Dockerfile and script in <https://github.com/justwriteclick/versions-jekyll>.

Additional uses of CI

You can also use continuous integration (CI) to integrate with a translation server for docs. Within OpenStack, a translation team uses the translation server—currently Zanata—to translate manuals. Whenever a change is merged, the CI infrastructure server uploads the current text automatically to the translation server, so that translators can directly translate and always have the latest strings. Once a day, a so-called *periodic* job is run on the CI infrastructure to download all translated strings from the translation server to the docs repos, and a change is proposed with any new strings. The documentation team has a chance to run the translation import through the CI infrastructure together with a manual review.

Additionally, you can use the CI infrastructure to synchronize any shared files from one repo to a few others. For example, multiple repos can share a single glossary, together with translations of that glossary, thereby gaining efficiencies in reuse and preventing translation rework. OpenStack automation manages shared documentation by checking whether files in other repos need an update after a change set, and then running a script that proposes a change. This type of configuration enables humans to run the test scripts for a final review.

Summary

We hope that this section gives you ideas about how you can use CI/CD for docs processes, including shared content and translated content. We have found the benefits far outweigh any risks in the approach. Our need to match with other teams means we adopted the continuous mentality to ship early and often. Look at your docs, open source or otherwise, with an eye towards automation, and see what obvious solutions appear.

Andreas Jaeger

Andreas Jaeger and Anne Gentle wrote the CI/CD sections collaboratively. Andreas and Anne are both members of the OpenStack docs team, and Andreas is also involved with the OpenStack CI Infrastructure. He has contributed to various open-source projects for over 20 years and works for SUSE.

Review and test content

Content review fits well with quality assurance goals. Learn about the tools to use to review docs, which are the same tools that are used to review code.

One aspect of docs reviews is running automatic quality tests. See the “Automate testing” section for more information about automation.

The open-source Gerrit review system, which runs on GitHub or another system, can be another piece of your review system. GitHub pull requests provide the review mechanism, while Gerrit provides code collaboration and a way to approve or reject changes.

Finally, use a docs review checklist to ensure that your reviews are thorough and consistent.

This section covers the GitHub and Gerrit review systems. Both systems are integrated with Git version control.

- [GitHub reviews](#)
- [Link to bugs in docs patches](#)
- [Measure improvements](#)
- [Write down review expectations](#)
- [Gerrit reviews](#)
- [OpenStack example: Gerrit reviews](#)
- [Automate testing](#)
- [Prioritize technical reviews](#)

- Coach better contributors
- Create and fix doc issues
- Deliver copy edits
- Summary

GitHub reviews

In 2016, GitHub added a set of user interface workflows that enable you to review patches. You can still do line-by-line comments with the **Add single comment** button. What's additional is the ability to comment on the overall patch and either request further changes or approve the changes. There's also a setting that lets administrators on the repo require that all pull requests must be approved before merging. Read the GitHub documentation for [reviewing changes in pull requests](#).

You can assign reviewers to a Pull Request (PR) and also ignore reviews if needed. Get agreement from your team about what review steps are necessary for merging a pull request. You can also read more about [permissions for GitHub](#) when merging or closing pull requests. User accounts and organization accounts have finely tuned permissions.

Link to bugs in docs patches

As a best practice when working with docs contributors, let reviewers know the reason for your patch by referring to the bug or issue that the patch fixes. Such a reference helps reviewers compare the fix to the issue and judge whether the fix solves the stated problem.

For example, OpenStack has a shortcut system that lets a contributor put phrases like `Closes-Bug:#nnnnnn` in the commit message, where `#nnnnnn` is the bug number in Launchpad (the bug tracker used by OpenStack). This link between a doc bug and the contribution makes it easier to tell whether the patch resolves the bug. Reviewers can click the link for the bug and read the comments in it.

GitHub has the same system. You prefix the GitHub issue number with a pound sign (`#`) in your commit message or a review comment, and GitHub automatically creates a link to the relevant issue.

Ensure that contributors can tell whether a bug has been accepted. In OpenStack, the `confirmed` status indicates that a bug was accepted. To accomplish this in GitHub, add labels to issues. Then, when you add an issue link to your pull request and the pull request is merged, the issue is automatically labeled `closed` . Provide label descriptions in your contributor guide to help people use labels as intended.

Measure improvements

By having documentation metrics, you can look for areas to improve to meet certain goals, such as adding contributors or decreasing doc complaints or errors. One great feature of GitHub is that it provides metrics for contributions and for issues. You can visualize and measure improvements in the documentation itself, or you can focus on improving processes that enable more contributors. For example, [Rackspace went from 6 documentation contributors to 175 contributors](#) in less than a year after adopting a docs-like-code model.

Write down review expectations

Let your reviewers know what you expect in a review. Do you want to release quickly with technically accurate docs? Or, do you want to publish the docs on a specific release schedule? Do code patches merge only when the docs are up to par with the code?

Document your review expectations in the contributor guide for both new contributors and seasoned reviewers.

A [review checklist like this one available from a docslikecode.com newsletter](#) can be a helpful tool to encourage thorough and consistent reviews. Categories provide groupings of questions to consider while you review, and the answer to the first question could prevent further review. For example:

- ☐ Do I have the configuration in place to test the instructions?
- ☐ Have I run all the commands in the doc changes and matched the results?

Use your team's priorities, tests, and style guide to modify the following examples to create your team's review checklist:

- ☐ Does the document build without errors?
- ☐ Is the output formatted as expected?
- ☐ Are the headings correct for the style guidance and overall organization?

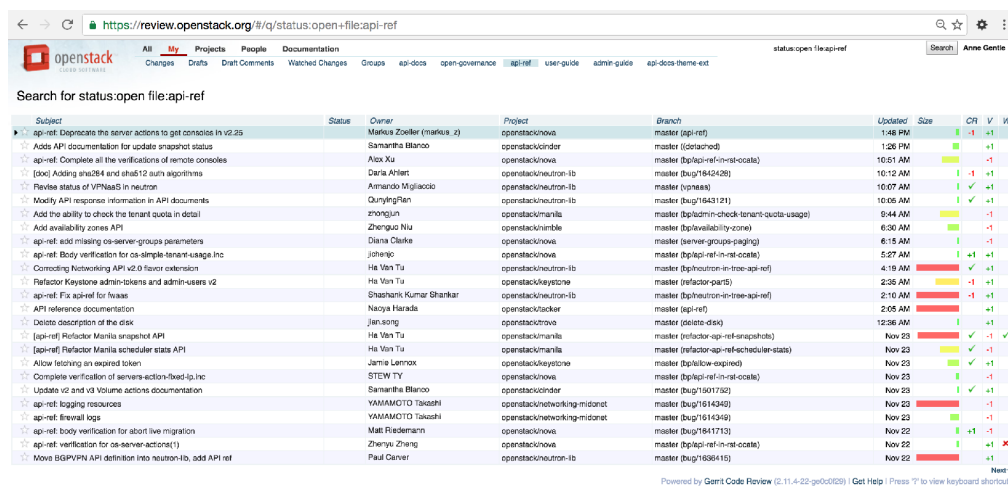
- `[]` Will the audience understand the context and why this information matters to them?
- `[]` Has data been properly redacted if necessary for security reasons?
- `[]` Are screenshots accurate for the version?

In GitHub, the `- []` markup displays a cleared check box when you view it online, and the `- [x]` markup displays a check mark in the check box. Read [Task lists in all markdown documents](#) on the GitHub blog for details.

Gerrit reviews

Gerrit is a web-based collaboration tool that enables teams to write code together, and it can be used in the same way to review doc patches written in a source file format. It integrates closely with Git.

Gerrit provides a configurable *dashboard* that enables you to review across multiple repos by using search keywords and status indicators. For example, you can search for all API doc changes across dozens of repos by searching for specific file extensions in specific folders. Consistency across repos is necessary so that the search can locate all the relevant files that you would want to review.



The screenshot shows the Gerrit web interface with a search bar at the top containing 'status:open file:api-ref'. Below the search bar is a table of search results. The table has columns for Subject, Status, Owner, Project, Branch, Updated, Size, CR, and V. The results list various API-related changes across different OpenStack projects like openstack/nova, openstack/neutron-lb, and openstack/keystone.

Subject	Status	Owner	Project	Branch	Updated	Size	CR	V
api-ref: Deprecate the server actions to get console in v2.35	Open	Markus Zoeller (markus_z)	openstack/nova	master (api-ref)	1:48 PM		-1	+1
api-ref: Add API documentation for update snapshot status	Open	Samantha Blanco	openstack/linder	master (detached)	1:26 PM			+1
api-ref: Complete all the verifications of remote consoles	Open	Alex Xu	openstack/nova	master (api-ref-in-tree-cosole)	10:51 AM			-1
[doc] Adding and24 and sha12 sum algorithms	Open	Daria Akhmet	openstack/neutron-lb	master (bug1642428)	10:12 AM		-1	+1
Revised status of VIFaaS in neutron	Open	Armando Migliaccio	openstack/neutron-lb	master (typenaa)	10:07 AM			+1
Modify API response information in API documents	Open	QunyingRen	openstack/neutron-lb	master (bug1643121)	10:06 AM			+1
Add the ability to check the tenant quota in detail	Open	zhonglun	openstack/manila	master (api-admin-check-tenant-quota-usage)	9:44 AM			-1
Add availability zones API	Open	Zhenguo Liu	openstack/nimble	master (api-availability-zone)	6:30 AM			-1
api-ref: add missing os-server-groups parameters	Open	Diana Clarke	openstack/nova	master (server-groups-paging)	6:15 AM			-1
api-ref: Body verification for os-simple-tenant-capsa-inc	Open	jiuhong	openstack/nova	master (api-ref-in-tree-capsa)	5:57 AM		+1	+1
Correcting Networking API v2.0 flavor extension	Open	Ha Van Tu	openstack/neutron-lb	master (api-ref-in-tree-api-ref)	4:18 AM			+1
Refactor Keystone admin-tokens and admin-users v2	Open	Ha Van Tu	openstack/keystone	master (refactor-part5)	2:38 AM		-1	+1
api-ref: Fix api-ref for fwaas	Open	Shashank Kumar Shankar	openstack/neutron-lb	master (api-ref-in-tree-api-ref)	2:10 AM		-1	+1
API reference documentation	Open	Naoya Harada	openstack/racker	master (api-ref)	2:05 AM			+1
Delete description of the disk	Open	Jian Song	openstack/crow	master (delete-disk)	12:36 AM			+1
[api-ref] Refactor Manila snapshot API	Open	Ha Van Tu	openstack/manila	master (refactor-api-ref-snapshot)	Nov 23			+1
[api-ref] Refactor Manila scheduler stats API	Open	Ha Van Tu	openstack/manila	master (refactor-api-ref-scheduler-stats)	Nov 23			-1
Allow letting an expired token	Open	Jamie Lemoor	openstack/keystone	master (api-ref-expired)	Nov 23			+1
Complete verification of servers-action-fixed-ip-inc	Open	STEVE TY	openstack/nova	master (api-ref-in-tree-capsa)	Nov 23			-1
Update v2 and v3 Volume actions documentation	Open	Samantha Blanco	openstack/linder	master (bug1601732)	Nov 23			+1
api-ref: logging resources	Open	YAMAMOTO Takeshi	openstack/networking-midonet	master (bug1614349)	Nov 23			-1
api-ref: Firewall logs	Open	YAMAMOTO Takeshi	openstack/networking-midonet	master (bug1614349)	Nov 23			-1
api-ref: body verification for abort live migration	Open	Matt Rickmann	openstack/nova	master (bug1641713)	Nov 22			+1
api-ref: verification for os-server-action1	Open	Zhenyi Zhang	openstack/nova	master (api-ref-in-tree-capsa)	Nov 22			+1
Nova BGPVPN API definition into neutron-lb, add API ref	Open	Paul Cinner	openstack/neutron-lb	master (bug1636415)	Nov 22			+1

Gerrit also has a concept of *teams*, which have the ability to merge a change by voting.

OpenStack example: Gerrit reviews

To make changes to OpenStack code and docs repos, you use the Gerrit code review system. The OpenStack infrastructure team runs Gerrit, which is a web-based code collaboration and review tool. While pull requests provide the review mechanism in GitHub, Gerrit provides the review interface in OpenStack.

The basic workflow is that a docs contributor clones the docs repo, makes changes to the documents, tests them locally, commits them to Git, and then uploads them to OpenStack's Gerrit instance. Gerrit then sends a change notification to the continuous integration (CI) services for software development.

After the notification from Gerrit arrives, the CI system runs the various tests that are configured for the repo. In reality, OpenStack runs multiple CI instances in parallel and uses a home-grown tool called Zuul to coordinate. You can see the status of any given build on the [Zuul website](#).

After the change is uploaded to Gerrit, reviewers can see it and comment on it. The Gerrit web user interface enables line-by-line reviews. So, a reviewer can comment directly on any problems spotted in the source file. The tests also build the docs, so a reviewer can see built docs in HTML or PDF when applicable.

Comments on the patch can state what's wrong with it, ask questions for clarification, or state that the patch is fine. These comments help the original author and other reviewers update and evaluate the patch.

After reviewers comment on a patch, they can optionally vote for or against the change. Voting is an evaluation of whether or not the patch can be implemented. A positive vote means that the patch can be implemented; a negative vote means that the patch needs more work. A reviewer can also abstain from voting and provide only a comment.

All reviewers can register their vote in the OpenStack Gerrit tool:

- `0` . No score.
- `+1` . Looks good to me, but someone else must approve.
- `-1` . This patch needs further work before it can be merged.

A special group of senior reviewers, or *core reviewers*, can also vote and approve a patch so that it can be published. Core reviewers can register the following votes:

- `+2` . Looks good to me (core reviewer).
- `-2` . Do not merge.

After two core reviewers vote `+2` , a core reviewer—normally the second one who gave a `+2` to the patch—approves the patch and then it is merged and published. A

patch with negative review comments does not get approved, so the docs aren't published until consensus is reached with the proper approvals.



The screenshot shows a code review interface. At the top, there is a 'Reply...' button on the left, and 'Patch Sets (6/6)' and 'Download' dropdown menus on the right, followed by a star icon. Below this is a large, empty text area for comments. At the bottom, there is a voting section. It includes a row of radio buttons for scores: -2, -1, 0, +1, and +2. Below these are two sections: 'Code-Review' and 'Workflow'. The 'Code-Review' section has radio buttons for -2, -1, 0 (which is selected), +1, and +2, followed by a 'No score' option. The 'Workflow' section has radio buttons for -2, -1, 0 (which is selected), and +1, followed by a 'Ready for reviews' option. At the very bottom, there are two blue buttons: 'Post' on the left and 'Cancel' on the right.

The automatic testing also results in a vote during the review phase. When a change is approved, the system runs the tests again—on the changes merged with the then current, updated Git repo—to ensure that a merge did not introduce a failed build. A change can merge only if the test system reviews the change positively.

Each test job starts a virtual machine (VM). The VM checks out the repo with the tested change, installs any dependencies for the test suite, and runs the test suite. Several public clouds made available to the OpenStack project, currently by Rackspace, OSIC, OVH, Bluebox, and Internap, run these VMs to automatically test all changes, not just doc changes. In early 2017, the OpenStack centralized infrastructure used up to 1900 VMs to run automated tests. Yes, OpenStack teams use the cloud to produce docs about the cloud.

Automate testing

When you treat docs like code, you can make incremental changes over time that ensure high-quality and technically accurate docs. You can also follow practices that will help to ensure the quality of the content. Automated testing, for example, is valuable, although it can be difficult. You must also find ways to incorporate editing best practices in your quality controls and provide valuable insights. Don't risk alienating contributors by focusing on minor errors rather than valuing the contributor's technical contribution.

Which tests should you run? What value do these tests provide?

Of course you want to ensure that the docs build. With a build tool such as Jekyll (Ruby) or Sphinx (Python), you can incorporate other automated tests.

Automated tests are either voting or non-voting tests:

- **Voting tests.** These tests block a patch from going through the gate and merging. Voting tests save humans the time of downloading a patch locally and running tests manually.
- **Non-voting tests.** These tests report but do not block a patch from going through the gate. For example, *Are the translations still working with this patch?*

We recommend that your builds run these automated tests:

Voting	
File syntax	<p>Is the language syntax correct in all files in the patch?</p> <p>This test checks the syntax of individual files and helps contributors quickly locate syntax errors.</p>
Docs build	<p>Do the docs in the patch build?</p> <p>This test checks the status of docs builds. Optionally, you can upload the built docs to a draft server so that reviewers can easily review the newly generated content to see how a change looks in HTML and PDF.</p>
Links	<p>Do all links work?</p> <p>This test checks whether links to external websites work.</p> <p>You can also mark this test as non-voting to accommodate the possibility of off-line external websites.</p>
Deleted files	<p>Does the patch delete any files that are used by other deliverables?</p> <p>This test checks that all docs still build if one file is removed.</p>
Non-voting	
Translated docs build	<p>Do the translated docs in the patch build?</p> <p>This test checks the status of translated docs built through the tool chain.</p>

Niceness	<p>Is the content in the files <i>nice</i>?</p> <p>This test checks for extraneous whitespace, overly long lines, some unwanted Unicode characters, or proper formatting (JSON). Extra whitespace and overly long lines make it more difficult to review source files side-by-side. Instead of looking at a lot of red or green lines that identify white spaces, you can focus your reviews on content changes.</p>
----------	--

For additional efficiency, run these tests only if they are relevant to the patch. For example, if the patch does not delete any files, do not run the deleted files test.

OpenStack test scripts

For automated testing examples with the tox testing tool and shell or Python scripts, see [openstack-doc-tools](#) in the OpenStack repo.

In OpenStack, the centralized infrastructure enables execution of scripts, and the docs team has its own repo with test scripts, mostly written in Python.

Developers made those scripts by using the same workflow used for docs. After major changes, the test tools are marked for release, and that release is then used to test any docs changes. In the docs repos, there's a Python convention of a `test-requirements.txt` file that indicates which release version of `openstack-doc-tools` works with a given set of doc source files.

To enable reviewers to concentrate on content and not quibble about form, the automatic tests handle most of the details. However, the team does not require all the automatic tests to pass to publish the document.

There are a few optimizations in the test scripts. For example, when DocBook XML file builds became expensive, a small dependency builder checked which files changed and which guides included those files, and then ran builds only for those guides. Other tests, like syntax or URL checks, are run only on the changed files. These optimizations are currently not needed on RST files, because RST files are much easier to parse and building the guides is faster.

The tests do not run any grammar or spell checkers because a voting check must always be accurate.

There are plenty of discussions about automating spell checks, but such checks really require a human for judgment calls.

Prioritize technical reviews

The docs-as-code ecosystem values technical accuracy above all else and relies on reviewers to guarantee it.

It's hard to predict how long it will take to verify the technical accuracy in a docs patch. However, it is critical that a human double-check the technical accuracy of any docs contribution. To save time, you can set up environments that test user actions as part of a review.

OpenStack example test environments

Following are some examples of tools that help set up test environments for documenting OpenStack. These tools let you work like a developer, using the same tools that they do.

DevStack	A collection of scripts that let you run a development environment that is configured to a local setup.
TryStack	A free public cloud run with donated hardware and resources. A TryStack account gives you free cloud resources through CLI or API commands.
Paid cloud accounts	Paid public or private cloud accounts enable you to test API calls.

Coach better contributors

You can use reviews to coach your contributors' writing skills. Use comments in your contributors' pull requests to demonstrate good writing, suggest organizational changes, and instruct them in the finer (or the completely nonsensical) points of the English language.

Some people who speak English as a second language are fascinated with and enjoy writing in English. Others are shamed when they misuse English, so they are reluctant to write docs. The tech docs world is often English-centric, yet plenty of tech writers write first in French, German, or Japanese and then translate their content to English.

Writers might hesitate to work with others on deliverables, and developers might feel they don't have much to contribute to the docs. But in the docs-like-code world, no one person knows everything, and you can achieve better docs through collaborative writing and reviewing.

Some guidelines:

- Let newer reviewers do the reviews on the first day that a patch is up. This approach gives new reviewers more time to learn the doc set and complete their reviews.
- Encourage more experienced reviewers to step in after the initial review. This approach lets newer reviewers learn from other reviewers' comments in subsequent

reviews.

- Pair experienced reviewers with newer reviewers. To tag additional reviewers, use the @ symbol with a GitHub user name in a review comment. For example, `@annegentle` .
- For small technical suggestions, patch the patch yourself. Never merge a patch, however, until the original author has a chance to review your changes.
- If your review might sound a bit harsh to a new reviewer, reach out on chat, IRC, or in a private email. Let contributors know that your aim is to instruct and that you appreciate their work.

Anne's experience with the review balancing act

At an internal developer conference at Rackspace in 2015, Anne wanted to know what other teams do about review quality—specifically, how to improve the review experience from both sides. For example, she disliked marking only trivial mistakes or stylistic differences to the conventions.

Everyone would rather do technical reviews and ensure consistency and quality, of course, but with time constraints, Anne had to find out how to get better at prioritizing what she reviewed.

At this developer conference, people suggested a short waiting period. Yes. Wait to review and let others review first. Also, they suggested that contributors should patch other people's patches. Ensure that the original author still agrees, but make the docs experience easier and even enjoyable.

Create and fix doc issues

To notify others about doc bugs, you can create issues. For example, if you spot a typo in the docs, you can create an issue. After the docs team verifies that the issue is a true issue, you or someone else can submit a pull request to fix it.

For example, at the end of the calendar year, you might update the copyright year in the docs in response to a doc issue.

If the docs team uses labels effectively, you can click a label to find all related issues. For example, the Bootstrap docs repo uses a [docs](#) label.

To find content to update, look for unassigned issues in the repo. If you can assign the issue to yourself, do it! Otherwise, comment on the issue to request that someone assign the issue to you. Then, begin work on the issue.

For large projects, the usual workflow is to fork the repo and issue a pull request. To find out how to make updates, look for a contributor guide or a `CONTRIBUTING` file in the repo.

Ideally, the contributor guide tells you how to make updates. For example, see [Contributing to Bootstrap](#).

For excellent examples of GitHub Pages, see [GitHub Pages examples](#). Peruse these repos to find existing source docs to update.

Deliver copy edits

If you have worked with a tech editor, you know that copy edits can be numerous and complex. Back when editors used a red pen to mark up printed-out pages, writers complained that editors made their content bleed!

Traditionally, the editor provides markup to a writer who updates the source files. However, when you work in GitHub with developer contributors, this method has limits. GitHub doesn't have a built-in way for editors to mark up changes in a file. And developers might not be willing to or have the time to make tons of editorial changes.

Writers and editors can use a couple of techniques to deliver copy edits to other writers and to developers. The technique that you use depends on whether contributors can incorporate editorial feedback and on the number of comments.

- If contributors can incorporate edits and you have a small number of comments, enter line comments in a pull request. For numerous comments, copy the content to an app such as Microsoft Word or Google Docs and then mark it up. Word has easy-to-use track changes and commenting features that enable an editor to show changed content, ask questions, and make comments. Google Docs has similar features. Then, create a PDF from the marked-up doc and attach that PDF to a comment in the pull request or to a separate issue. The

original author can then make updates in one or more pull requests, or the team can triage the issue to verify the editor's suggested changes.

- If contributors can't incorporate edits, you can update the file itself and create a pull request with your changes. When you create the pull request, you can enter line comments with additional questions or suggestions. The contributor can compare the differences between the original and changed content and decide whether to merge the pull request as-is or make updates before merging.

When contributors make their own updates, they can see their mistakes more clearly, learn about style and writing guidelines, and become better contributors. However, workloads might not permit contributors to take the extra time to make updates. So the key is flexibility.

Professional technical publishing of books on GitHub

Technical publisher O'Reilly has used Git repos for years when working with technical authors on content. For example, when working with the production team at O'Reilly on a line-by-line copy-edit of the entire OpenStack Operations Guide, the team had multiple writers at the ready, able to enter their edits. The production team emailed the exact process. Their quality control process had both a copy-edit stage and copy-edit review stage, where someone verified that all suggested edits were made. Here's an excerpt from their acceptance email when the manuscript went to production:

The purpose of the copy edit is to check for errors in structure, language, grammar, spelling, and formatting. The copy editor reads the entire manuscript and marks any errors. The copy edit is sometimes done in batches (chapter by chapter), if appropriate.

Summary

Reviews can be done partly by automatic tests, and many reviews are completed by people using review systems. Ensure that you understand the permissions for merging and closing pull requests, and then look for ways to guide reviewers. Setting expectations for reviewers and linking to doc bugs in the commit messages help reviewers get better at reviewing documentation. A review checklist is also a great way to indicate top priorities and completeness to reviewers.

Publish docs

Generally, you publish the product docs when developers release the software.

You could publish your docs more often than the developers release their code but when you treat docs like code, the source files for your docs often use the same version control system that the code uses.

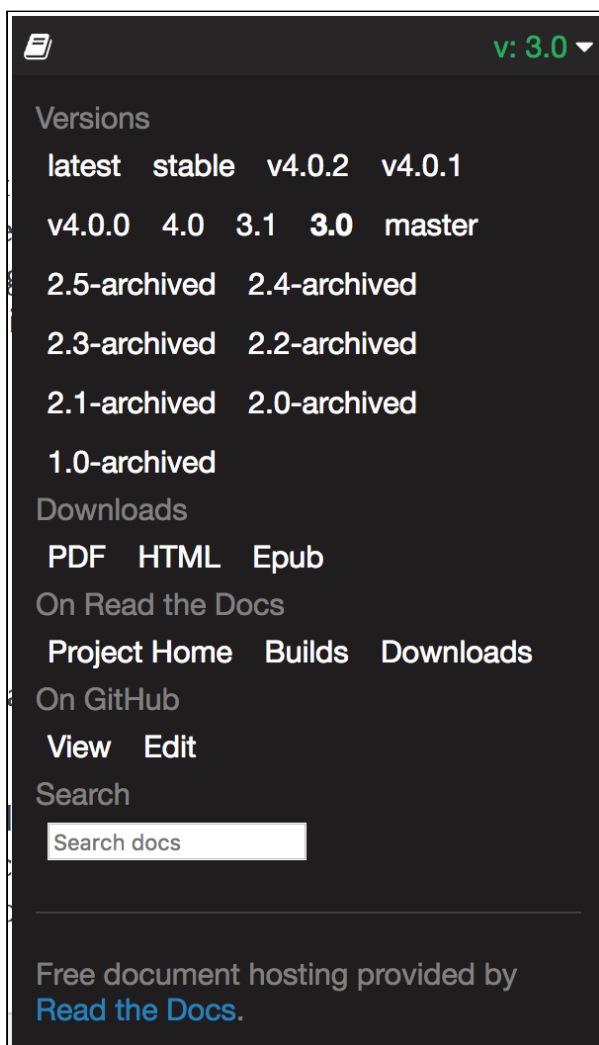
You can exploit the features of this version control system for your docs. For example, when a reader views a back-level page, a banner can appear that links to the latest version of the docs.

The docs URL can also indicate the docs version, such as `/latest/` or `/4.1.3/` . Or, the URL can indicate the release status, such as `stable` or `beta` , or the docs branch, such as `master` or `develop` .

- [Define a docs version](#)
- [Define a docs locale](#)
- [Notes on translations and continuous integration](#)
- [Docs source version control](#)
- [Use case: How Read the Docs supports versions](#)
- [Use case: How OpenStack docs use versions, branches, and tags](#)
- [Releases for documentation sites](#)
- [Summary for publish and release docs](#)

Define a docs version

The [Write the Docs](#) Sphinx theme is recognized for its excellent version badge design. On the site, a small version badge floats on each page. To choose another docs version or download another output format, the reader simply clicks an arrow control.



To implement this version badge, you specify the version number in a parameter in a `conf.py` file with each build. Maintenance is straightforward and you can integrate this

parameter with the version control system.

The [Django documentation site](#) also implements a version badge.



You can also use the docs version parameter to:

- Show a banner that lets readers navigate to the latest or different versions of the docs.
- Use the URL to show the docs version, release status, and docs branch.

For example, a `/latest/stable/master/my-docs` URL indicates that the docs are a stable release of the latest docs from the `master` branch.

Other things to consider:

An unsupported version is no longer available online

OpenStack redirects requests for unsupported versions to a newer version. However, that strategy sometimes results in outdated content. Because OpenStack stores source files in a public location, it includes instructions for how to get and build the docs for the unsupported version locally for personal use.

Because your projects and organizations might have different legal requirements for unsupported doc requests, plan ahead to accommodate those needs.

A page exists for one but not another release

For example, when the reader views a new page for the v1.2 release and then switches to an earlier version of the docs, the new page does not exist in the earlier version. To handle this situation, you can create a page that describes the reasons that the reader should upgrade to the v1.2 release.

Your static site uses a single landing page to reflect multiple releases

Maybe the product manager wants to highlight a new feature or alternatively wants to show consistency release-over-release on the single landing page for the site. You must determine what to show on your landing page. Make sure that casual readers who search for a certain release find a landing page that addresses that release.

You must retract a version for security reasons

Do you simply redirect the reader to a later or earlier version? The URL to which you redirect the reader depends on which version your service uses instead of the retracted version.

A service skips a semantic version

What if your release skips from v4.1.1 to v4.1.4? Do you need to explain why v4.1.2 and v4.1.3 are missing?

Define a docs locale

You can also use a configuration parameter to define when a translation is available for a particular version. When you indicate a docs version, you might also consider displaying the language, even if you do not have translated content available. It pairs nicely with versions and lets your design grow into further release models if needed. So think ahead about your URL containing `/de/latest/` or `/es/2.4.23/` and a language indicator on the page with the version indicator to let people know which version they are reading and if it's available in another language.

Notes on translations and continuous integration

Many open source projects provide tightly-coupled integration with translation teams from around the world. Since the beginning of the OpenStack project, technologists who speak English as a second language have put forth great scripts to help with managing translations even when the docs site could not stop producing content and did not enact a doc freeze, only a string freeze for interfaces. When the typical cycle for documentation is to freeze the English version so that the other language versions can be released as well, this is a supreme feat. The automation that enables the system to have scripted imports and exports while enabling human translators rather than machines is truly impressive. Read more about it in the [OpenStack I18n Guide](#).

The OpenStack translation team's goal is to:

Incorporate translated strings into a new release so that more global users experience a translated version of OpenStack.

While the docs usually start with written English, some open source communities write a document in their native language first and then provide translations to English. This model

provides successful documentation to multiple users and emphasizes that English is not the only language for technical content.

Docs source version control

Will you use tags, branches, or releases to provide version control for your documentation source? When you choose one or even multiple methods, how do you use tags, branches, or releases to automate publication and provide a release of documentation? In some cases, you are tied to the publication methods, whether pushing to an external website that has a version-control system itself, or producing flat files that must reside in a particular directory structure.

Realize that the complexity increases as you add requirements for protecting source access. When you also must protect output access, you have multiplied the complexity of your solution. While open source projects can work on the docs in the open while drafting, many software projects must write and review docs with a limited group of collaborators, such as in a private repo. This requirement may increase the cost of continuous integrations, for example. When you also must protect draft copies of published documentation, you need a staging server or a login system.

Each possible solution for the requirement increases in complexity as you share and reveal less and less during both development and production. Development costs also increase. Be ready to consider the trade offs as you analyze.

Versions, tags, and branches with Git for docs source

When using Git, you can apply a tag to a particular commit by pointing to that point in the history of the repository. A lightweight tag is a pointer to a specific commit, but isn't really used for releases, as you likely want more information about a tag in order to release the docs. An annotated tag has the checksum, a label, and information about who did the tagging and on which date.

To tag a repository at a particular point in the history, refer to [Git Basics - Tagging](#). When you tag locally, you also push those tags so that other collaborators can see them when cloning or pulling from the repo.

Versions in GitHub are based on Git tags. You can use the GitHub web interface to [create your repository's releases](#). When you do so, you can download a compressed file of all the source files in the repo at that moment in time. You can base a version on any branch, and tags are used to indicate to contributors the version number of the doc source files. Because of the limits of pairing tags with a compressed file, and because tags must be re-pushed if changed, it's generally not a best practice to "move" a tag once it's already on contributor's local environments.

For strict release environments, where a new release would be required if the tag changed, you may not want to set expectations of pairing tags with releases even though GitHub does. Generally though, it's best to use the same system as the

overarching software and choose the system with some familiarity for contributors or readers who tend to read doc source files rather than output content.

Use case: How Read the Docs supports versions

More than 50,000 projects use the Read the Docs infrastructure system to automate their documentation builds with publishing. Their [versioning implementation](#) is based on the version control system and the default branch used there. For example, Git-based version control often uses `master` as the default branch, but you can change that label to the `develop` branch depending on your desired workflow.

Use case: How OpenStack docs use versions, branches, and tags

For OpenStack, the docs team uses tags and stable branches to version and archive doc source files. The team has a [documented process for releasing the docs site](#) prior to a coordinated, timed release of multiple OpenStack projects at one point in time, twice a year. The team uses tags on the common docs repo, marking a point-in-time for the docs that went with a particular release of OpenStack. The team uses `stable/releasename` branches for the supported releases of OpenStack that have common docs stored in the `openstack-manuals` repo. The team has documented [instructions for creating the stable branch](#).

When someone needs to read the docs at a point-in-time for an unsupported release, they can get a [copy of the tagged repository and build from those files](#).

When a contributor wants to continue to publish changes to a past release that is still supported, they back-port changes to the `stable/releasename` branch and publish the docs site again.

Both of these tasks, tagging and branching for release, are well-documented so the release tasks are shared by team members. Ideally the need for a cherry-pick and back-port happens rarely, and the team does not need to move the `git`

`commit` value on a tag for an end-of-life release. Then, when someone must build docs from an end-of-life release, he or she can follow instructions to do so.

Releases for documentation sites

Because web-based documentation sites are often released once they are publicly available, you might need to pre-plan in stages to ensure the release of a docs site goes smoothly.

Similar to an online service, you should start to think of your docs site in terms of testing, staging, and production. A testing area for doc builds helps you try out different designs, test and review patches to the docs, or try test processes and workflows. A staging area enables writers to see what their output will look like when published, as a staging area should be as identical to production as possible. You could consider a staging area to be like a backstage of a play. Only certain readers can have access to a staging area for a docs site.

Also consider that your doc tools should be released and finalized prior to a final release of a documentation set. What I mean by this is that you should be able to replicate the exact output from the tool as the tool existed at the point-in-time of a release. As OpenStack migrated from Maven builds with DocBook source to Sphinx builds with RST source, we needed to ensure we could release the Maven build tool prior to needing to build the DocBook to HTML and PDF files that looked like they did at release time. Again, these considerations may have different legal or contractual requirements in your organization.

Archives contain all files for a release

When you create a version in GitHub, you are likely creating an archive of the source files. If you also want to archive the built files, you must check built files into a branch and create a version that contains the built files as an archive. Generally speaking, repos do not contain compiled or built files, so you must complete this additional configuration step to create an archive for docs.

Examples of releasing documentation

OpenStack releases documentation with two patterns that are well-established in the adjacent software projects within OpenStack. One release pattern is a continuous publication pattern of the website, which is not coordinated with other projects, known as `independent`. Some of the OpenStack documentation is written this way, such as User Guides, when not much changes during release-to-release of the services. Another is `cycle-with-milestones`, where the document needs to be published at pre-determined times in order to synchronize with a project's release. The Configuration Reference is an example of this doc release model in the OpenStack sphere, because new configuration options are introduced and only available at a certain released point of the software. It's important to ensure the page documenting those configuration options clearly indicates in which release you can use that option. Also, that document must clearly show which options have been deprecated in a given release.

For a service that has new features every six weeks or so, you must provide release notes in addition to new pages that describe how to use the new features.

When reading a page about a feature your service didn't have yet, you need a way to get to the page relevant to the product you have installed. Consider both the reader's experience and the contributor's experience when releasing a documentation site or other deliverable. Find an example written as a proof-of-concept using the static site generator, Jekyll, at <https://github.com/justwriteclick/versions-jekyll> .

This example demonstrates using version control for source and output, with output directories defining the version in the output. The HTML and CSS and JavaScript files can also work in concert to provide the user experience for selecting and reading another version, stored in the output directories.

Summary for publish and release docs

Because of the different perspectives on both source and deliverables for documentation, you must think about versions, archives, and possibly translations when treating docs like code. Also consider the two perspectives on versions: the contributor's views and the reader's views of the version value. Each version implementation can differ slightly depending on the consumer of the version number. As you reveal content throughout the process, the amount of control you need over the source, the output, or both, increases the complexity of the solution.

TUTORIAL: GET STARTED WITH DOCS LIKE CODE

To help you get started with docs like code, this tutorial shows you how to *fork*, or make a copy of, the [Beautiful Jekyll](#) GitHub repo. This repo is a “ready-to-use template to help you create a Jekyll website quickly.” Then, use your fork to create a website that you host directly on GitHub through [GitHub Pages](#).

At first, these steps can appear challenging. Keep an open mind as you dive into unfamiliar authentication systems and command-line instructions. It’s not that you don’t know this; you just don’t know this *yet*. Much like learning a musical instrument, you must practice. Zed Shaw, author of [Learn Python the Hard Way](#) and other programming books, reminds us that the hard way is not hard but is daily practice.

To embrace a development mind set, we recommend that you use the command line. That said, you can also use the GitHub user interface (UI) to try out GitHub. Learn with the tools that feel comfortable. No judgment here, only learning.

After you have some practice, look for doc repos that need editing or have doc issues listed for contributors to work on. Go to the [GitHub Guides site](#) for more tutorials.

Task overview

1.	Learn Git concepts .
2.	(Optional) Install Git .
3.	Fork a repo .
4.	Configure Jekyll (GitHub UI) or Configure Jekyll (command line) .
5.	View your website.
6.	Make edits to a repo .
7.	Troubleshoot Git , if necessary.

All methods use the [forking workflow](#).

Install Git (optional)

If you plan to use the command line to configure Jekyll and make version control changes instead of using the GitHub web editor, install Git for your operating system.

Mac OS X

When using Mac OS X, run commands in a Terminal window. There are several ways to install Git for Mac. For version 10.9 (Mavericks) or later, running `git` at the command line prompts you to install Xcode Command Line Tools, which installs Git. You can also install using Ruby and Homebrew:

1. In a Terminal window, install [Homebrew](#).

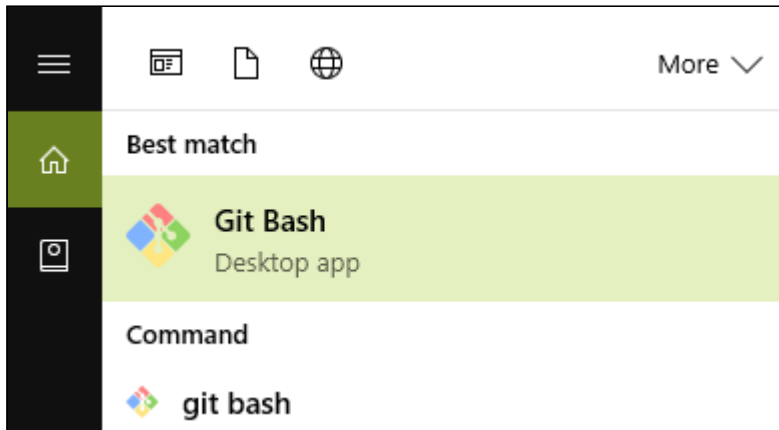
```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/H
omebrew/install/master/install)"
```

2. Use Homebrew to install Git:

```
$ brew install git
```

Windows

When using Windows, run commands in Git Bash, which is a BASH emulation used to run Git from the command line on Windows. The [Git for Windows installer](#) also installs Git Bash.



Linux

When using Ubuntu or Debian, install with the `apt` package manager.

```
$ sudo apt-get update
$ sudo apt-get install git
```

When using Fedora or other RedHat-related distributions, install with the `yum` or `dnf` package managers.

```
$ sudo dnf install git
```

Or:

```
$ sudo yum install git
```

Also follow instructions for [authenticating with GitHub](#) from [Git](#).

Fork a repo

To use GitHub to edit and host your website, get a GitHub account and fork, or copy, a repo to your user name.

1. [Create a GitHub account](#).
2. [Authenticate your account](#).

If you intend to automate, set up SSH keys. A practical familiarity with SSH keys helps with automation.

3. [Fork the Beautiful Jekyll](#) repo.
4. On [GitHub.com](#), log in and navigate to your `beautiful-jekyll` fork. Typically this URL is `https://github.com/USERNAME/beautiful-jekyll`.
5. Click the **Settings** icon.
6. Go to **GitHub Pages** and under **Source**, choose **Master branch**.
7. Click **Save**.

Next, configure Jekyll.

You can edit the configuration file from the command line or through the GitHub UI.

Configure Jekyll (GitHub UI)

To edit your website in the GitHub web editor, follow these steps.

1. Navigate to the `_config.yml` file for your forked repository in your web browser and click **Edit**. Typically this URL is `https://github.com/USERNAME/beautiful-jekyll/blob/master/_config.yml`.
2. On the **Edit file** tab, set `url` to `USERNAME.github.io` and set `baseurl` to `/beautiful-jekyll`.
3. To commit your changes to the `_config.yml` file directly to the `master` branch, select **Commit directly to the master branch** and then click **Commit Changes**.

Configure Jekyll (command line)

To edit your website in `vi` using the command line, follow these steps.

1. Open Git Bash as an administrator on Windows by right-clicking the command in the **Start** menu, or open a Terminal window on Mac or Linux.
2. Type `git --version` to make sure Git is installed. If you get an error, install Git for your operating system.
3. Use one of the following methods (SSH or HTTPS) to clone your fork, where `USERNAME` is your GitHub user name:

SSH	<pre>\$ git clone git@github.com:USERNAME/beautiful-jekyll.git</pre>
HTTPS	<pre>\$ git clone https://github.com/USERNAME/beautiful-jekyll.git</pre>

4. Change to the `beautiful-jekyll` directory:

```
$ cd beautiful-jekyll
```

5. Set up and fetch the `upstream` remote:


```
$ git remote add upstream
https://github.com/daattali/beautiful-jekyll
$ git fetch upstream
```

You are currently on the `master` branch.

6. Open the configuration file by using the `vi` text editor:

```
$ vi _config.yml
```

- a. To enter insert mode, type `i` .
- b. Set `url` to `USERNAME.github.io` .
- c. Set `baseurl` to `beautiful-jekyll` . For example:

```
# If you are building a GitHub
project page then use these
settings:
url:
"http://annegentle.github.io/bea
utiful-jekyll"
baseurl: "/beautiful-jekyll"
```

- d. To exit insert mode, press `ESC` .
- e. To save and exit the file, type `:wq` and press `RETURN` .

In the command:

:	Moves the cursor to the bottom of the screen.
w	Saves the file.
q	Quits the file.
RETURN	Executes the command.

For information about `vi` commands, see [A quick reference list of vi editor commands](#).

7. Commit your changes and push the `master` branch to your fork:

```
$ git commit -a -m "Initial commit  
for website"  
$ git push origin master
```

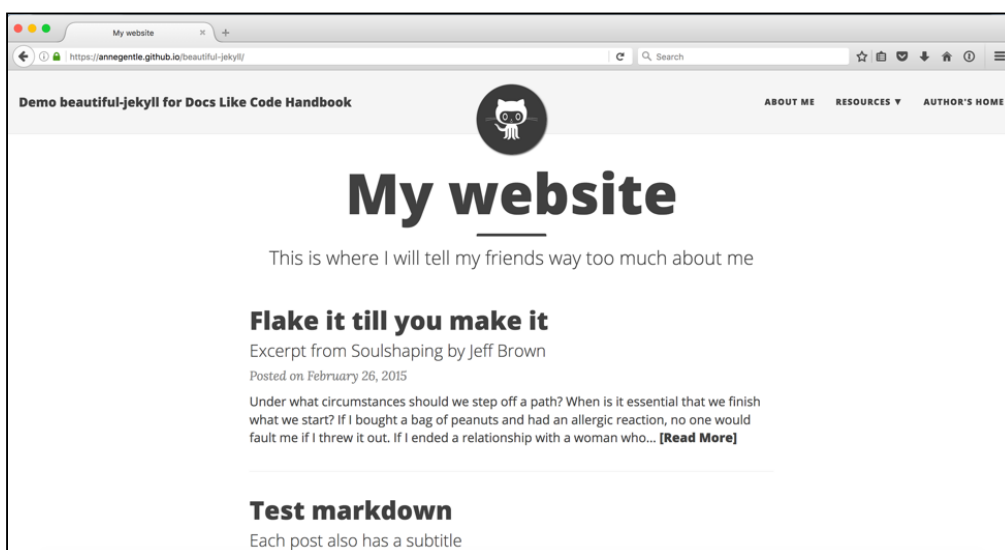
For git errors, see [Troubleshoot Git](#).

View your website

Celebrate! You used GitHub to publish a website.

In a browser, navigate to

`http://USERNAME.github.io/beautiful-jekyll` to view your website:



If your site does not build, GitHub sends you a “Page build failure” email message with an explanation and link to a help article.

Make edits to a repo

This example shows you how to make edits to the Bootstrap repo by using the forking workflow.

Rorschach PR bot

With Bootstrap, the Rorschach PR bot automatically checks all pull requests. If you get a build problem comment on your PR, read the comment, fix the problem locally, and push the revised version to your PR's branch.

In this example, run the commands in either Git Bash on Windows or Terminal in Mac OS X or Linux.

1. To fork the [Bootstrap project](#) repo, click **Fork**.
2. Clone your fork of the repo:

```
$ git clone  
https://github.com/drfleming0227/bo  
otstrap.git
```

3. Change to the `bootstrap` directory:

```
$ cd bootstrap/
```

4. Set up and fetch the `upstream` remote:

```
$ git remote add upstream  
https://github.com/twbs/bootstrap  
$ git fetch upstream
```

You are currently on the `v4-dev` branch.

5. View your remote labels:

```
$ git remote -v
```

The command shows your remotes:

```
origin  
https://github.com/drflaming0227/bootstrap.git (fetch)  
origin  
https://github.com/drflaming0227/bootstrap.git (push)  
upstream  
https://github.com/twbs/bootstrap (fetch)  
upstream  
https://github.com/twbs/bootstrap (push)
```

6. Check out a `my-v4-dev` branch that is based on the `v4-dev` branch:

```
$ git checkout upstream/v4-dev -b  
my-v4-dev
```

7. Make edits. To keep your change set small and easy to review, work on a single file at a time, if possible.
8. Commit your changes and push the `my-v4-dev` branch to your fork:

```
$ git commit -a -m "Initial commit  
to my-v4-dev"  
$ git push origin my-v4-dev
```

9. Test that your changes build correctly. See [Running documentation locally](#).
 10. On GitHub.com, [open a PR](#) with a commit message that follows the upstream guidelines.
-

Troubleshoot Git

Because Git has its roots in Linux kernel development and is intended for software development, its error messages are not very friendly. To troubleshoot Git, you must learn how to interpret these messages.

Common errors include repo connection problems, authorization or identity problems, and confusion about commit messages. When you configure your system to use `git` commands, errors can occur when you enter something that Git does not expect. Many configuration errors are related to the identity that you use to authenticate when you use `git` commands to connect to GitHub.

Helpful git commands

To troubleshoot your local configuration, run these commands:

<pre>\$ git remote -v</pre>	<p>Lists the <code>remote</code> labels and URLs for <code>git</code> remotes.</p> <p>You configure <code>git</code> remotes by repo. To determine whether the URL for either upstream or origin is a fork, look at both the labels and the corresponding URLs.</p>
<pre>\$ git config -- list</pre>	<p>Lists your local configuration for the repo.</p> <p>Lists the user name, email address, remote labels, and information about the <code>master</code> branch. Use this command to ensure that your authentication credentials represent who you think you are for that repo.</p>
<pre>\$ git -- version</pre>	<p>Lists your <code>git</code> client version.</p> <p>Sometimes the <code>git</code> client changes the default settings so a particular command might not work as it used to. To determine whether you are affected by a client change, use this command, the release notes, and the help information returned at the command line.</p>
<pre>\$ git status</pre>	<p>Indicates the current branch and status of local files.</p> <p>Lists the local branch and which files are staged for commit, not staged, or untracked. Use this command to determine whether your commit contains the files and changes you intend.</p>

You can define a global configuration or a repo configuration. Be sure that you know which configuration is being applied when you see an error. See [Basic Client Configuration](#).

Error: Permission denied (publickey)

This error occurs when your identity does not meet GitHub's expectations. This error has various causes. For solutions for the Mac, Linux, and Windows platforms, see the GitHub article, [Error: Permission denied \(publickey\)](#).

To troubleshoot configuration errors, verify these steps:

- Did you connect to the correct server? For example, are you using GitHub.com or an Enterprise GitHub server?
- Did you use the correct `git` user when you connected through SSH?
- Did you use the correct SSH key?

To troubleshoot SSH and public key errors, see the GitHub article, [Categories / SSH](#).

Error: Repository not found

This error occurs when you run a `git clone` or `git push` command with a misspelled remote name. This error can also occur when the GitHub user that you have configured for local use does not have permissions. To troubleshoot this error, see the GitHub article, [Error: Repository not found](#).

Fatal: HTTP request failed

This error sounds a little scary but it typically indicates a configuration error or a two-factor authentication error, such as when you enter an incorrect passcode. When you enable two-factor authentication, you must provide a personal access token instead of entering your password for HTTPS Git. This failure can surprise you if you thought you set up your local remote for SSH access and find that the remote URL starts with `https` instead of `git@`.

Merge conflicts

A common error is a *merge conflict*:

```
CONFLICT (content): Merge conflict in
filename.rst
Automatic merge failed; fix conflicts
and then commit the result.
```

To troubleshoot and resolve merge conflicts:

1. List files that are in conflict in the working directory:

```
$ git status
```

For information, see [git-status](#).

2. After you edit the affected files to resolve the merge conflicts, add and commit your changed files:

```
$ git add path/to/filename.rst
$ git commit -m "<Informative
commit message>"
```

“Could not apply...” errors

Sometimes you must *rebase*, or change a series of commits, to combine commits, change commit messages, or delete commits that are no longer necessary. When Git can’t merge

two changes, it makes notes inside the files, and a human needs to look at the files to determine what parts to merge together. When this happens, the error message looks like this:

```
error: could not apply 05fc552... Adds
serialized response
formats to End User Guide
When you have resolved this problem,
run "git rebase --
continue".
If you prefer to skip this patch, run
"git rebase --skip"
instead.
To check out the original branch and
stop rebasing, run
"git rebase --abort".
Could not apply
05fc55210a580dffacb72cf65988f979d9870f
a8...
Adds serialized response formats to
End User Guide
```

Fortunately, Git gives you the commands that you need in the text that it returns on the command line.

1. To determine which files Git could not apply changes to, run:

```
$ git status
```

2. In the messages, look for the `both modified` line:

```
both modified: cli_swift_howto.rst
```

3. Open the affected files one at a time and look for conflict markers within the files:

```
<<<<<<< HEAD
```

The lines marked with `HEAD` are the lines in the base branch. The lines marked with `>>>>>>>` include the SHA value for the commit in which these lines were committed.

4. Decide which changed lines to keep and which changed lines to delete, and delete those lines.
5. Delete the lines with the conflict markers, including the `<` and `>` symbols.
6. Save and close your editor.
7. To tell Git that you've made the changes, type the following command in the Terminal window:

```
$ git add filename.rst
```

8. Commit your changes and then push the changes to the appropriate remote:

```
$ git commit -a -m "All conflicts  
are resolved"  
$ git push origin <branch-name>
```

LESSONS LEARNED WITH DOCS LIKE CODE

People wonder whether treating docs like code can work in their environment. Although outcomes matter and this movement aims to exceed users' expectations, the disruptions in your team's daily tasks, work expectations, and areas of control can make this transition difficult.

For example, think about what goes through employees' minds when they face a new system:

“Can this scale?”

“Can my team maintain the tooling required?”

“What do you mean I have to review 10 patches before getting to write myself?”

“I’m a programmer, not a writer; why do I have to add docs to my task list?”

“Work was easier when I could send it to the tech writers to handle.”

“I am the owner of the configuration docs. Why would I let others work on something I’ve controlled for years? They might mess it up.”

As this guide has shown, the shift to treating docs like code includes a complex overhaul of attitudes, processes, toolsets, and expectations. However, many teams work through these difficulties to great rewards. To ease the transition to using more docs-like-code techniques, look for these opportunities:

- [Create a great web experience](#)
- [Equip your contributors with a style guide](#)
- [Empower your contributors](#)
- [Write a contributor's guide](#)
- [Build in continuous integration for docs](#)
- [Teach everyone to respect the docs](#)
- [Test and measure outcomes](#)
- [Summary](#)

Create a great web experience

Your contributors want to know where the content goes. For them, communicate the layout of the site and use meaningful directory- and file-naming conventions.

For readers, provide excellent navigation systems and ways to find what they need.

Because traditional tech pubs toolsets aren't good at creating wonderful, responsive websites that provide an amazing docs experience, modernize your toolset. A natural tension exists between producing artisan, hand-crafted docs and producing tens of thousands of web pages that still offer a great experience.

Equip your contributors with a style guide

To build a trusted set of great reviewers, you must give them a style guide to use while writing or reviewing.

Unless you establish standards for terminology, your contributors end up arguing about which name lands in the source. For example, is it *plugin* or *plug-in*? A style guide frees your contributors to focus on what's most valuable: technical reviews. Instead of calling out grammatical errors or capitalization consistency issues, reviewers can point writers to the style guide.

A style guide will also be helpful when you create tools to automate quality checks.

Empower your contributors

When you engage a large number of contributors, many of whom you don't personally manage and direct, you can lose control of what gets written. You don't get to pick what is written first, second, or third, or even whether something is written at all.

Create a culture where contributors are empowered to prioritize work: let them determine what to work on first, second, and third. Be ready to trade control for many other benefits. And just be sure to have processes and systems in place to triage docs issues.

Write a contributor's guide

Most code repos use a README file to explain how the contents work. When you are writing documentation in a source code repo, use the README file to explain exactly how someone can contribute, to display build status, and to describe any automation tools and how to use those locally to test changes.

You can also add a CONTRIBUTING file to the root of the repo to [describe guidelines for repository contributors](#).

Describe what contributors can expect at review time, or how to give their patch its best chance in landing in the source, or how to log an issue. When a contributor uses the GitHub web UI to create an issue or pull request, they see a link to the CONTRIBUTING file in the web UI.

You can also provide templates and examples of the best work you have ever seen, and point contributors to your style guide.

The 18F digital services agency at the United States General Services Administration provides the [18F Content Guide](#), which is a good example of a content guide. Their [18F Open Source Style Guide](#) helps people document code repositories.

Which comes first, the style guide or the contributor guide?

In the early days of OpenStack, there were only two projects and one technical writer, Anne. She started on the wiki, doing a content audit of what existed, and wrote a quick style guide that was a single page on the wiki. The essence of this style guide was simple: write in the active voice, use initial capitalization for titles, and value technical accuracy above all else. Eventually, as more contributors began working on the documentation, the wiki page grew into a few wiki pages. Fast-forward a few years, and the [OpenStack Documentation Contributor Guide](#) has grown into its own document and contains the style guide. It describes markup conventions, tooling, what content goes where, and has a section dedicated to API documentation.

Build in continuous integration for docs

Automate builds and quality checks so that you spend your time delivering great content, not running builds. Let the robots perform the builds, and use tools that don't require a seat license so that the writers don't have to build from their computers only. For example, in OpenStack, the team can merge 50 changes a day in a single repo that contains 10 deliverables (web-delivered guides). The infrastructure builds draft copies with scripts for people to review online.

To succeed at automation, use tools that enable automation, like static site generators. If you are a tech pubs tools product manager, your next big features are Travis CI job integration, Jenkins job implementation, and any scripted way to build your output so that humans don't have to click anything to build the docs.

When you must make changes across multiple repositories and deliverables, be ready to learn bash scripting or get help with it. Fortunately, text manipulation is pretty powerful at the command line. Unfortunately, some team members or outside contributors might be using non-Linux-based systems and have no tools to help with this problem.

Teach everyone to respect the docs

Depending on your contributor base, you influence more people to respect the docs and encourage more contributors to improve the docs when the docs are in a code system. For example, across multiple repositories, OpenStack delivers REST API docs for almost 30 services. Some days there are over 100 changes that must be reviewed for REST API docs. This amount is intimidating, and the toughest part of the job can be disappointing people by not reviewing their changes in a timely manner.

Also, consider teaching developers to review code patches that also contain docs changes. Encourage them to merge code only after the docs are up to par with the code changes.

Test and measure outcomes

Much of the techniques and technology are still in early stages of a software lifecycle, so do not cement yourself into particular workflows or tools. Be ready to change a process if it's simply not working well for the team. Test the outcomes in two ways: with web analytics for the success of the output, and with contributor analytics for the success of the source creation and maintenance.

OpenStack has identified the best reviewers and coached people to become better reviewers by running a monthly metrics report and studying the results with data analysis as a group. When someone observes data that shows a new contributor only votes positively when reviewing, study the facts further on when they are voting and which patches they vote upon. Do they only vote when certain that another person has reviewed? Or are they voting only on subject areas where they have expertise? Or, do they need to gain confidence to be able to review a patch and reject it? These are metrics you can interpret about review patterns.

For contribution patterns, you can use the GitHub UI to look for metrics. For an example, look at the [Contributors graph](#) for the centralized docs repo for OpenStack, `openstack-manuals`

.

Look for a natural next step, using data queries to detect the best practices for docs-like-code techniques by analyzing contributor and reviewer data. Watch for opportunities to improve when you see best practices emerge based on data from large projects with historical data, or projects similar to your own. Team size, release frequency, commit frequency, ratios of docs and code metrics, and specialty areas related to docs and code collaboration are all interesting to study.

Summary

In some ways, this entire book is full of lessons learned. By stepping back and looking for best practices in code integration for docs, you can continuously improve your doc authoring, building, publishing, and consumption.

The main goal of treating docs like code is to fit in with the ecosystem, so that you can gain collaborators who know their stuff and keep the contribution tasks as simple as possible (but no simpler).

Glossary: Understand Git concepts

Familiarity with some Git terms and definitions is helpful as you consider a docs-like-code transformation.

branch

A parallel version of a repo within the repo that does not affect the primary or `master` branch. You can work freely in a branch without affecting the *live* version. After you make changes, you can merge your branch into the `master` branch to publish your changes.

New contributors can confuse named directories, such as a cloned fork that is named after the original repo, and Git branches.

You can instruct Git to base your branch on the `master` branch in upstream, origin, or another remote. For example, this command bases a new branch on the `master` branch in the `upstream` remote:

```
$ git checkout upstream/master -b  
<branch>
```

clone

A copy of a repo that lives on your computer instead of on a website's server.

commit

A point-in-time snapshot of a repo. Commits let you see the differences between changes. A commit is an individual change to a file or set of files. Every time that you save a file or a set of files, Git creates a unique ID, also known as the *SHA* or *hash*, that tracks the changes. Commits usually contain a commit message, which is a brief description of what changes were made.

downstream

A label for a remote URL, where a remote represents a place where code is stored. A downstream remote indicates an opposite of an upstream, or original, repo.

fork (noun)

A copy of the repo that is entirely yours in your namespace. A fork gives you a way to both contribute openly and get credit for your contributions.

fork (verb)

The act of making a forked copy of the repo.

issue

A way to submit a suggested improvement, defect, task, or feature request to a repo. In a public repo, anyone can create an issue. Each issue contains its own discussion forum. You can label an issue and assign it to a user.

organization

A collection of group-owned repositories.

pull request

A method of submitting edits that compares your changes with the original. Teams can view the comparison to decide whether they want to accept the changes.

push

Move your local committed changes to a remote location, such as [GitHub.com](https://github.com), so that other people can access them.

remote

A version of your project that is hosted on the Internet or on a network. The remote is usually connected to local clones so that you can sync changes.

repo

A collection of stored code or docs.

review

Perform a line-by-line comparison of a change and comment on improvements or suggest changes, much like a copy editor does for a newspaper article.

upstream

The primary label for the remote URL indicating the original repo where changes are merged. The branch, or fork, where you do your work is called *downstream*.