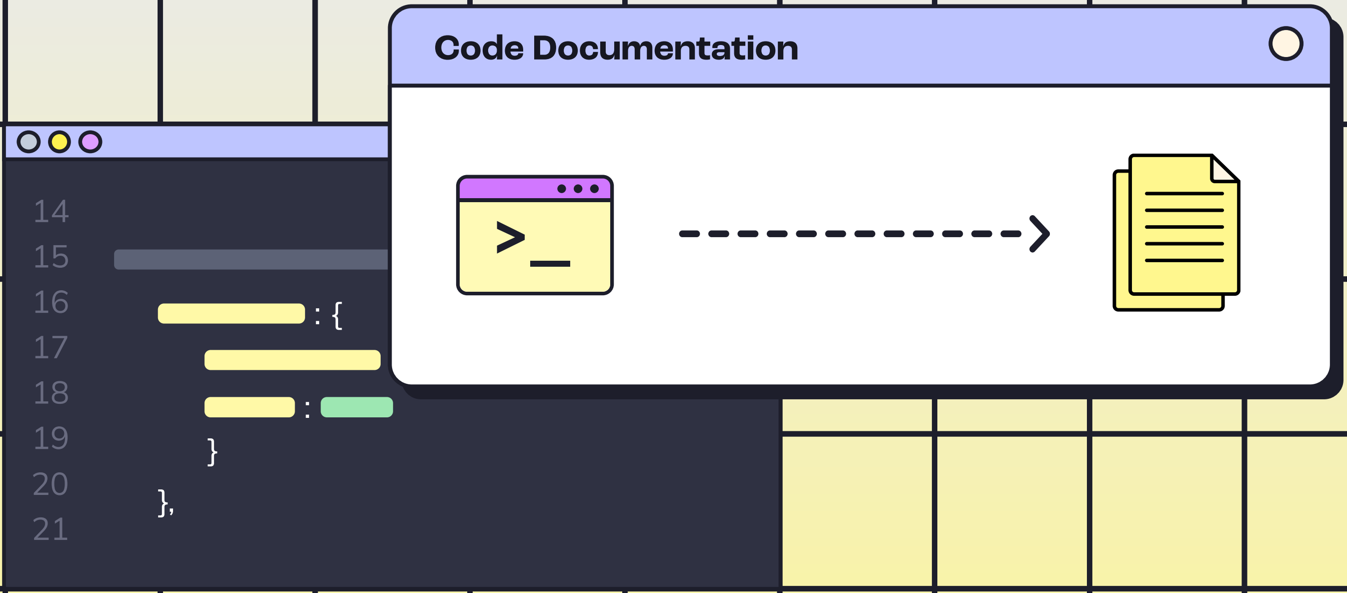
swim

The code documentation handbook

Best practices for effective documentation



Introduction

Developers and development teams both deserve and need good documentation. In fact, GitHub reports that developers are 50% more productive when documentation is accessible and available.

With effective documentation in place:

- New developers integrate seamlessly, reducing the learning curve.
- Teams remain on the same page, promoting agility.
- Knowledge isn't confined to a single individual; it's shared and available to all.

Yet, in today's fast-paced software development environment, a significant gap exists. Developers often grapple with outdated or insufficient documentation, and many lack the right tools or methods to create valuable docs. This not only affects their onboarding into new projects but also hampers their overall efficiency and the quality of the software they produce.

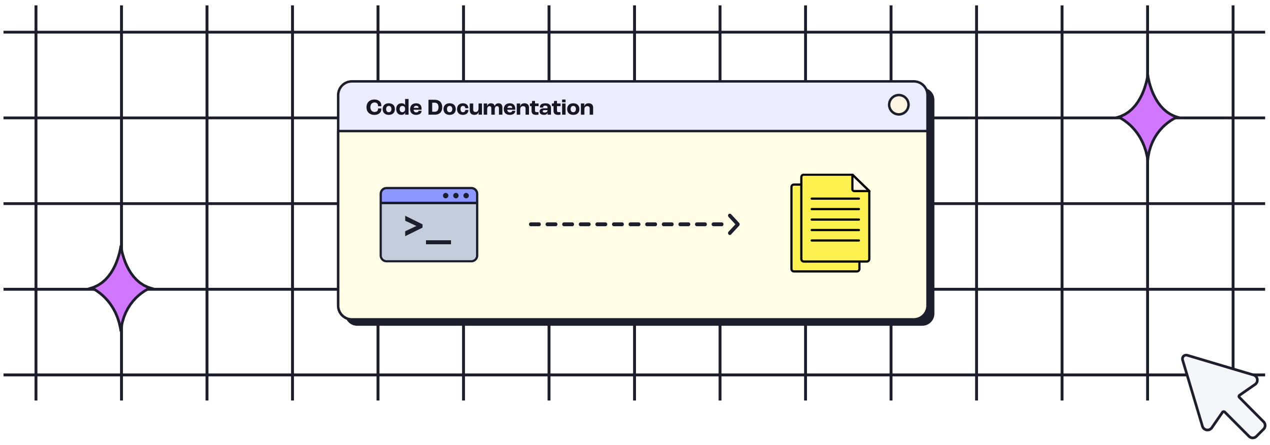
In this guide, we'll cover why it's important to document your codebase and will share practical strategies on how you can build a culture of documentation within your organization, ensuring that code knowledge is always within everyone's reach.

Table of Contents

Chapter 1

Why document your code?

We're also debunking the myths about **why** devs avoid documentation



Documentation is an investment worth making. Producing quality documentation takes time, but it's an investment that pays dividends time and again.

Documentation increases developer productivity

Despite being critical to onboarding new developers, improving code quality, and creating shared understanding for the whole team, documentation is often an under-resourced area of projects.

GitHub's research shows that easily sourced documentation translates to a 50% productivity boost for developers. Good documentation removes ambiguity and friction, getting the whole team on the same page.

Documentation contributes to building strong team cultures

GitHub's report went on to highlight the importance of good information flow in building trust and promoting developer satisfaction. Particularly in our current reality of geographically dispersed teams and remote working environments, documentation plays a critical role in ensuring that knowledge about your codebase flows among team members – both current and future.

Documentation is critical for onboarding new developers to the team

In research conducted by Harvard Business School, executives acknowledged that their companies could do better in onboarding new employees. In fact, a recent survey found that 52% of new hires feel undertrained following onboarding.

This perspective is confirmed by Gallup research of employees themselves, which found that only 12% of employees strongly agree that their organization does a great job of onboarding new employees.

Documentation contributes to building strong team cultures

GitHub's report went on to highlight the importance of good information flow in building trust and promoting developer satisfaction. Particularly in our current reality of geographically dispersed teams and remote working environments, documentation plays a critical role in ensuring that knowledge about your codebase flows among team members – both current and future.

Successful onboarding follows a two-step methodology:

1. Start with a high-level overview of your codebase. This step provides devs with the knowledge and information they need to get started.

Tip: *Use Swimm playlists. Playlists are an ordered sequence of documents and other resources.*

2. Learn as you go. Once devs have the full picture, they should be able to easily access the resources needed to get the job done.

Tip: *Swimm's IDE plugin eliminates the need to look for documentation. Discover documentation that already exists alongside code.*

Myths and misconceptions

Let’s be real for a second: most developers will admit that documenting code is one of the least enjoyable parts of the job.

This aversion is understandable, yet it often stems from a lack of appropriate tools used by teams to facilitate the creation of effective code docs. Let’s address and clarify the two most prevalent misconceptions regarding documentation:

Myth #1: Good code documents itself

By its nature, code has limitations in terms of its expressiveness. While it’s true that well-written code inherently demonstrates what it accomplishes, comprehensive documentation delves deeper, exemplifying the methods and rationale behind the code. It clarifies how the code achieves its purpose and, crucially, why specific coding approaches were chosen over others. Simply put, it focuses on the how and the why.

Relying on the code to explain itself fails to impart a full understanding of the underlying logic and nuances. Therefore, irrespective of code quality, it is vital that documentation fulfills this role, providing the necessary context and details.

While writing clean, concise, and readable code is highly beneficial and can reduce the amount of documentation required, it cannot replace the need for explicit documentation, especially in more complex and nuanced aspects of software development (think: complex logic and algorithms, large codebases, architecture changes).

Myth #2: There’s just no time

Creating code documentation isn’t just a time-consuming formality. It’s a strategic decision that amplifies long-term efficiency and success for development teams. Not only does good documentation ensure clarity and consistency in current projects, but it also provides clear guidelines for the future.

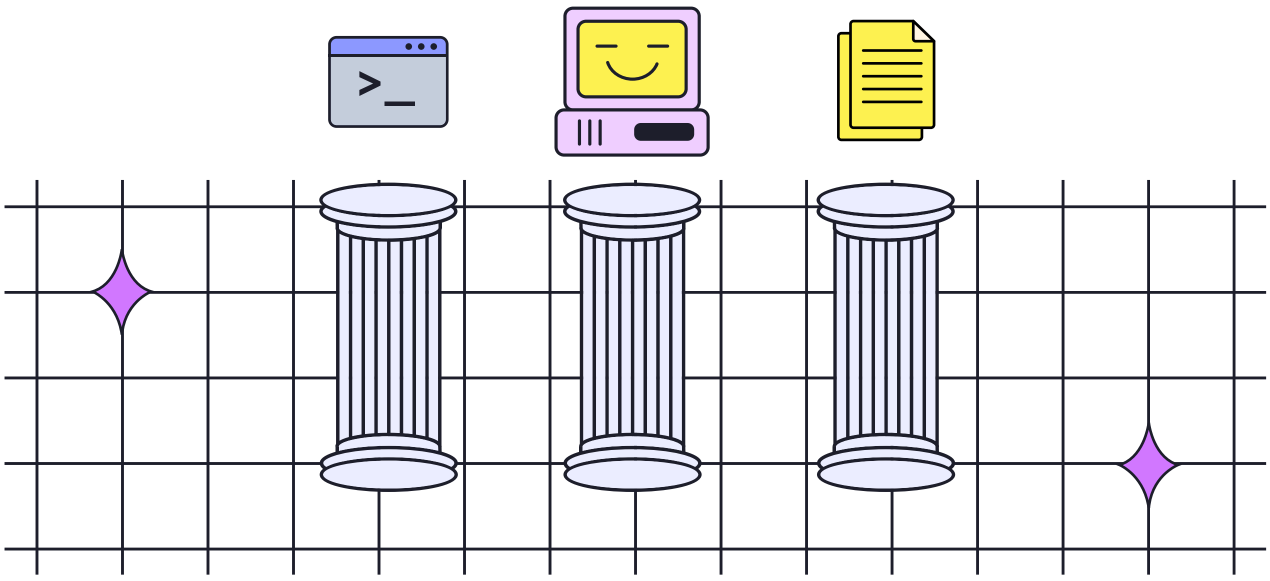
Proper documentation streamlines communication, aligning everyone's objectives and reducing misunderstandings. It safeguards knowledge, ensuring valuable insights aren't lost. And when developers spend less time deciphering complex code, they can focus more on innovation and coding itself, maximizing productivity and developer experience.

So while it might seem that allocating development hours to documentation at the expense of writing code is a diversion of resources, it's actually a strategic investment that yields substantial returns for the whole team.

Myth #3: You need to be a good writer to create effective documentation

You don't have to be the next Shakespeare to write good code documentation. The objective is to communicate information in a straightforward manner. An effective code doc focuses on the accuracy and comprehensiveness of technical details rather than on stylistic elements.

A deep understanding of the code and its underlying principles is key.
It facilitates the accurate conveyance of technical nuances and mitigates misunderstandings. The essence of effective documentation lies in clear, concise, and correct representation of technical information.
While good writing can be an asset, it is the clear and organized presentation of relevant information that defines the effectiveness of documentation in a technical context.



Chapter 2

Code documentation challenges

*

And how you can avoid them.

1. Code is non-linear

Not all code follows a step-by-step paradigm, and the assembly order is not always clear when documenting code. Things that appear at the top, such as variables, may relate to the functionality at the bottom. Functions defined at the end of the code could be executed in another block of code in the middle. This becomes more complex as code flows span across multiple repositories.

The core paradigm for writing documentation is a task-based approach, where you start with steps 1, 2, and 3 and continue until the task is complete. However, this model is not usually possible with code documentation because the code can be inherently non-linear. You cannot start from the top and go down—you might have to move back and forth between multiple files and repos.

2. Doc consumers have different levels of expertise

Another common challenge is deciding what details to explain and which to skip. Some developers might have a certain technical background and might be highly familiar with a certain framework or pattern, while others are not.

When documenting your code, it is important to base the documentation writing on the knowledge and requirements of your target audience, even at very different skill levels. If your audience has wildly different needs and awareness levels, you risk covering too much for experienced developers with explicit explanations or skipping important information for newer developers.

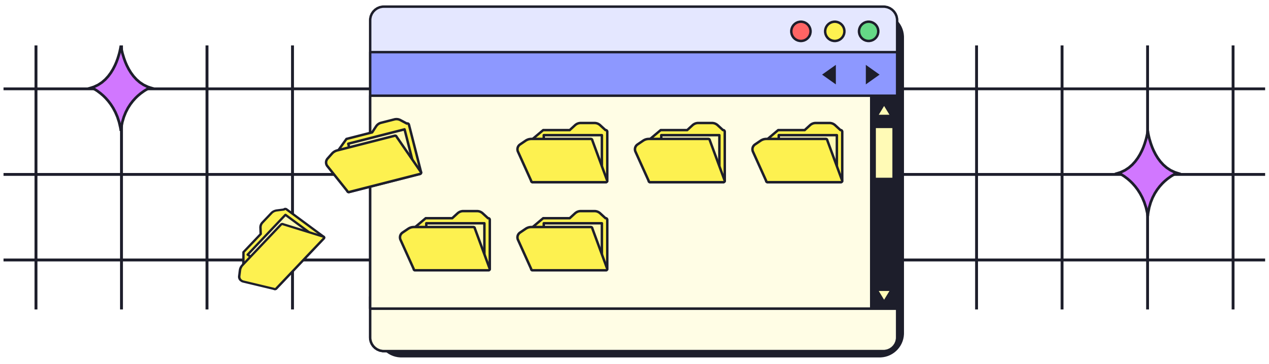
The solution is to break the documentation into separate documents for different audiences or provide an incremental discovery model, where initial information is provided first, and users can click through or expand to additional information based on their information needs.

3. Code documentation requires maintenance

Effective code documentation must include actual parts of the code it seeks to explain. You must ensure that the code samples work with multiple versions. For example, if you tweak a code sample, you have to go back and update the entire code in addition to your explanations for each section to keep everything in sync.

Documents can be difficult to maintain from one release to another if the documentation contains many code samples. You need to check that the code works. You might keep your code separate from the narrative context to test it more regularly.

Code examples are not the only case where documentation may become outdated. Any time any process related to the code is described, the code may change thus making the document obsolete. This can be true for specific details (such as numeric values), or high level concepts (such as the software architecture or main flows).



4. Documentation is hard to find

If developers can't easily access documentation when they need it, then what's the point of creating it in the first place?

Documentation should be readily available to developers, not something they have to search for. Consider this: while debugging or addressing critical issues, the last thing a developer needs is to waste time looking for relevant documentation. Its effectiveness is maximized when it's effortlessly accessible.

Swimm's IDE plugins integrate documentation directly within the development environment. Enabling developers to easily find, create, and maintain documentation, notifying them when existing documentation relates to a specific line or section of code they are working on. This means documentation comes to the developer, not the other way around.

Chapter 3

13 tips for writing technical documentation

✳

And actionable advice that actually works

1. Write it now. Refine it later

Your future self (and teammates) will thank you. A month later (even a day later), you won't remember the intricacies behind your code, or why you chose to create it in the way you did.

Write it now. Refine it later. Done is better than perfect.

2. Write it with a new developer in mind

Unless your development team is a one-person operation, developers will inevitably engage with code penned by their peers. Crafting documentation with new developers in mind guarantees uniform, precise information about the code and development workflow, diminishing the likelihood of misunderstandings and mistakes.

Additionally, clear, comprehensive documentation provides a foundational understanding, reducing the learning curve for developers who are newcomers to the team or are navigating unfamiliar segments of your codebase.

3. No need to explain every line of code

Document what you would need to be explained if you were picking up this code to work on it for the first time.

Write in short sentences and keep explanations concise and straightforward. We don't mean that you shouldn't write advanced technical code documentation. On the contrary, it's the elements that aren't so straightforward that need the best explanations.

4. Add references and code

Context is everything. Add code snippets (with markup language), libraries, API endpoints, parameters, coding conventions, and additional references to your explanations.

Referencing other sections of the documentation or code instead of repeating them makes your documentation easier to update. Maintaining these references makes it harder to keep your documentation up to date as the code changes. This is where Swimm's documentation tool makes all the difference – automatically detecting and marking where your document is out of sync with your code).

5. Add a quick start option

When onboarding a new team member to work with your code, you want to get them up to speed quickly. This is where a comprehensive collection of related resources can be invaluable.

Give your dev team a quick start page or bullet points that link to all the relevant resources.

The benefits:

- It significantly improves the user experience for your reader
- Provides context by giving them a high-level overview of the documentation
- Allows them to easily return to related resources when it's time to review them
- Makes the content easier to consume by breaking into chunks

To create a more sequenced and organized learning experience for new developers, take a look at Swimm Playlists, which allow you to group numerous types of related resources in a single location for your readers to work through in the order you choose. Swimm also has created documentation templates as a quick start option to help you get started.

6. Make sure it's accessible

Be sure to keep documentation in a centralized location, organized into logical folders or buckets. If you're using Word documents, PDFs, or Google Docs, ensure that you give viewing rights to the people who need to access them.

Of course, if you use Swimm's documentation platform, you'll find docs when you need them in your IDE and access them right next to the code they refer to.

If nobody can find or access your documentation, it begs the question as to the point of writing it in the first place.

7. Document code changes

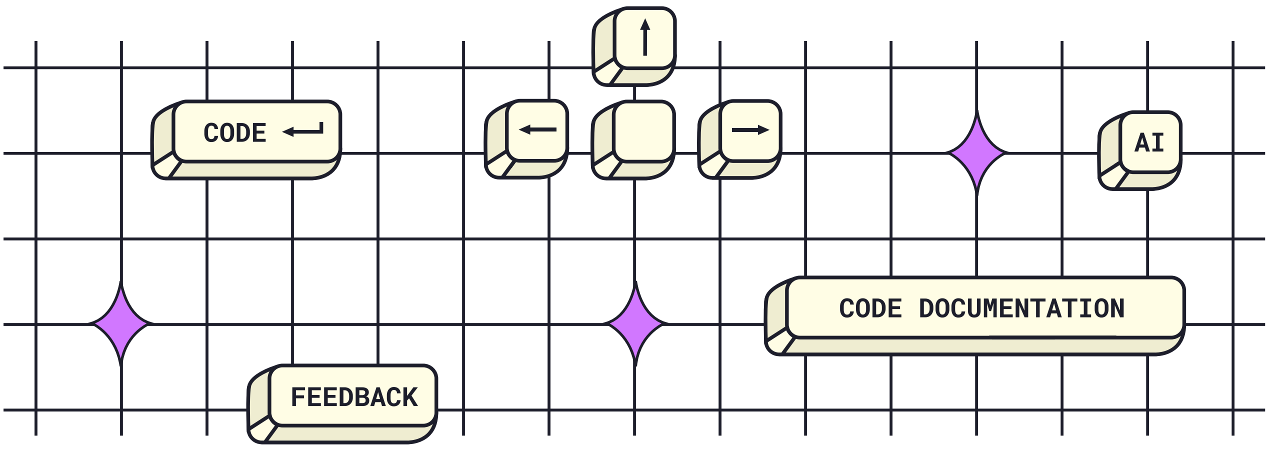
Document significant changes to your code, as you make these changes. By documenting when the changes occur, as part of your workflow, you make sure the information is fresh in your mind.

Swimm's AI capabilities enable you to automatically generate documentation based on changes made in a pull request.

8. Get feedback

Even if you don't have an "official" review workflow, share your documentation with your team and get their feedback.

A second set of eyes will help you recognize that what may be crystal clear to you might be not clear at all for another developer.



9. Create an updating plan

If you already have some documentation debt, it's time to jump in and get free of it. But the prospect can seem overwhelming. After all, the reason documentation is something devs prefer to avoid, usually has to do with the maintenance required.

Fortunately, there are documentation strategies to help you get moving in the right direction, along with Swimm's platform for facilitating ongoing documentation that's always up to date.

10. Make documentation as easy on yourself as possible

In order for documentation to accelerate the pace of development, it should be integrated into the development process, saving your team time - not adding to their workload.

Swimm automatically keeps documentation up to date as part of your CI flow. Easy, right?

11. Use glossaries

Keep your documentation streamlined by defining terms and using them as needed. And don't forget to link to the appropriate glossary entries when you use the terms you've defined.

12. Consider the medium

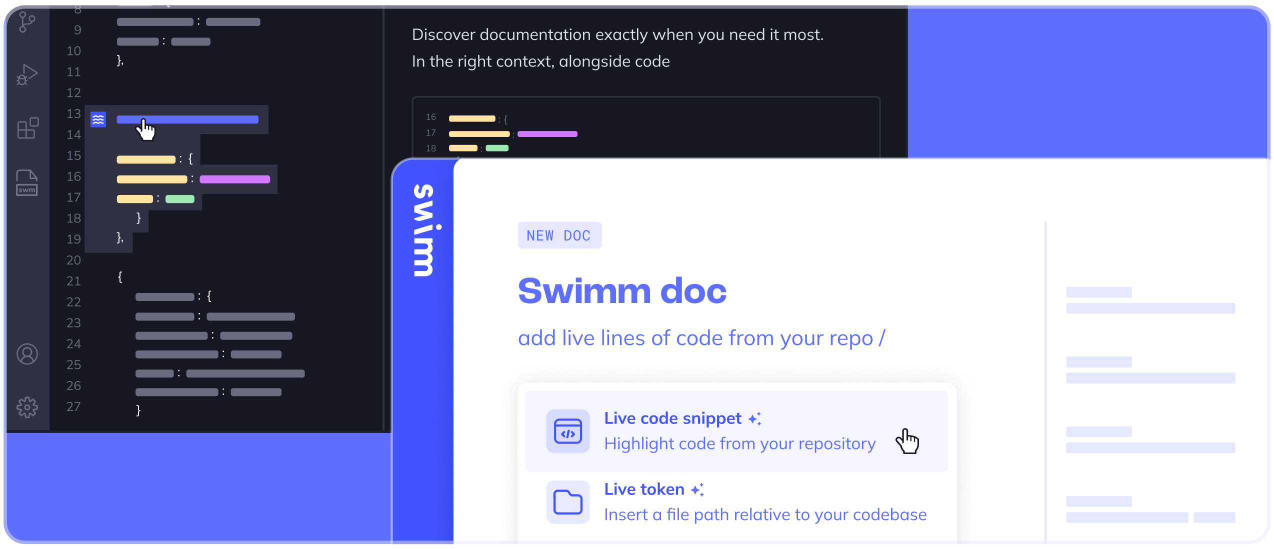
Great documentation consists of more than endless text. Consider the type of documentation that will get your point across best: annotated images and screenshots, code snippets, links to relevant outside articles, videos, diagrams, charts.

13. Use AI to get started

One of the most obvious use cases of Generative AI technology in the software development space is for code documentation. But, LLMs are only as powerful as the data they are trained upon. So while AI is powerful for helping with some aspects of code docs (think: suggesting an outline, generating specific explanations of code), LLM models are trained on the higher level details and nuances that code docs require.

Chapter 4

5 documentation tools you should know about


Discover documentation exactly when you need it most.
In the right context, alongside code

swimm
NEW DOC
Swimm doc
add live lines of code from your repo /
Live code snippet
Highlight code from your repository
Live token
Insert a file path relative to your codebase

Swimm

Swimm integrates code documentation into the development workflow, ensuring that documentation is always up to date and accessible when needed. Swimm was built by developers, for developers.

Key features include:

- Automatic synchronization of documents as the code changes
- Code coupled documents stored as Markdown (.md) within the codebase
- Powerful editing for software engineers, which makes it easy to embed elements from the code within the documents
- Easy access, discovery, and creation directly from the IDE via plugin
- Generative AI capabilities to structure documents and generate explanations of code snippets

Swimm integrates with the tech your team is already using, including GitHub, Gitlab, Atlassian, VS Code, JetBrains IDEs and so much more.

Notion

Notion is a flexible collaboration platform that provides a workspace where teams can customize pages, databases, and tables to suit unique requirements. It lets you lay out information creatively, giving your team full control of the documentation's display. Notion is suitable for organizations that want to use a single documentation tool but have multiple teams with varying needs. It also offers free personal plans.

Notion is known for its flexibility, which can also be a downside because it might not be as intuitive as a simple tool. However, creating static pages with text, embedded features, images, and tables, should be easy. Another drawback of Notion is that it does not include analytics capabilities.

Confluence

Confluence is an enterprise-grade documentation tool from Atlassian and is a convenient option for teams already using Jira or Trello. Confluence allows you to easily build and organize teams, promote collaboration between employees, and maintain documentation from any location. It is free for up to ten users.

Confluence emphasizes collaboration and simplifies communication within the platform. Users can comment, tag each other, share updates, and provide instant feedback to ensure all documentation is accurate and up-to-date. Confluence also offers built-in templates to make it easier to set up and implement documentation processes.

Nuclino

Nuclino is useful for organizing each team's information into a dedicated workspace. It allows you to create software documentation for customers and employees in public and private workspaces. Nuclino also offers features to make content more engaging, such as images, videos, tasks, embeds, and code blocks. You can use Markdown or the WYSIWYG editor to create content.

Nuclino enables real-time collaboration—team members can see changes as they write, eliminating the risk of conflict between versions. You can link to other pages in your knowledge base by typing @. You can organize items using workspaces and clusters. It has a commercial license with free and paid options. Nuclino's search bar lets you find relevant information, while graphs and boards help to visualize and organize content to help teams work intuitively. It integrates with many applications, including Google Drive, Slack, and Dropbox.

Gitbook

GitBook is a documentation tool for software teams that need to create private or public documentation. It is partly open source (with an Apache 2.0 license) with free and paid options. GitBook lets you synchronize your documentation with a GitLab or GitHub repository containing markdown files. However, this is a separate repository from the user's code, so making sure the documents are up-to-date is still done manually.

However, you don't have to use Git to use GitBook. You can also use the intuitive editor to create content or import content from other sources like Word and Markdown files. You can use unique knowledge bases called "spaces" and categories called "collections" to organize your content. Other important GitBook features include version control, real-time collaboration, co-editing, rich embeds, and simple PDF exports.

Chapter 4

Writing an effective code document in 5 steps with Swimm

Effective documentation helps other developers, technical staff and users achieve their goals. Our experience stems from documenting ourselves, and accompanying many engineers creating effective documentation. We have seen again and again how following a few guidelines can make a huge difference - both for creating more effective documentation and making the process easier for the writer.

This section assumes you know what document you want to write. If you are unsure, refer to the specific use cases section. To get started, sign up to Swimm.

Give your document an actionable title

Don't skip this step. It will help you focus.

When it makes sense, use one of these formats:

- "How to..." (e.g., "How to add a new Plugin")
- "How X works"(e.g., "How the recommendations engine works")
- "How we built X" (e.g., "How we built our CLI")

The title should tell the reader what they will learn from reading this doc.

Insert code snippets first, before writing any text

- Use the Swimm command /Code snippet before explaining why your code is important, focus on adding all code snippets that show the flow you are describing. If you are not describing a flow, select an example that demonstrates what you are describing, or any code snippet that might be relevant. If you are unsure what snippets you should add, it might be helpful to think of someone who doesn't know the code and highlight what you would show if you walked them through it.
- Make sure you add all relevant snippets.

Describe the snippets

Now that you have your snippets, you can re-order them (if necessary), and describe them. Pro tips:

- Explain why things are implemented the way they are.
- Focus on the information that's not in the code.
- Explain how a single snippet relates to the other snippets -its role in the flow.
- Refrain from explaining what exactly each line of code does. The code speaks for itself in isolation, and can tell parts of the story for you.
- Use smart tokens that are coupled to elements from the snippet or other locations in the code. Type backtick to search for code references in your repo and convert them to tokens.

Add an Introduction section

Explain what this doc is about in an introduction section.

When applicable, split into sections

Create sections using headings. They make the document easier to follow and navigate.

- In the Swimm web app, a Table of Contents navigation is automatically created on your right sidebar based on Markdown headings. Review it to make sure your structure is clear.
- It helps avoid writer's block. By focusing first on selecting code snippets, you don't bind your brain with copy or styling choices. When you get to actual writing, you'll already have a structure and code to prompt you, which is much better than a blank page.
- It makes you select a real example.
- The code speaks for itself, at least in isolation. By including these parts of the code, you don't need to explain them in English. You can then focus on parts of the story that are not clear within the code itself.

BUT WAIT

There's more!

Bonus: 2 templates to help you get started

Template 1:
Component or service overview

Introduction

This doc gives a high level overview of {{COMPONENT NAME}}. It is located under {{use `/path` to select the folder where the component / service is implemented}}.

Main features

The main features of {{COMPONENT NAME}} are:

Interface

{{How can this component/service be accessed?}}

Include an example from your codebase showing accessing this component/ service

<br/>

## Directory structure
{{Use `` `/path` `` and mention the main folders within this component/service}}

## Design decisions
{{Explain key design decisions}}

## Glossary
Here are some important terms to know:

Template 2:
Internal API

Introduction

In this doc we will describe the API for {{API Name (e.g., sending Analytic Events)}} and how to use it correctly. We use this API when {{use cases}}.

API definition

Select snippets of the various function's definition, so the reader can understand where the API is implemented

Simple usage

Show a simple example of using this API

Advanced usage: {{explain a scenario where this is needed}}

Show an advanced example of using this API

Best practices and additional notes

When using this API, it is important to follow a few best practices and avoid some common mistakes.

Show an example of a best practice and explain why it is important to implement the API this way swimm

Want to learn more about Swimm's reimagined approach to code documentation?

Sign up for a **community demo today** or

Get a personalized demo