
Software Technical Writing: A Guidebook

James (jamesg.blog)

Contents

License	5
Introduction	6
My experience starting as a technical writer	8
Why technical writing?	8
Mediums of writing	9
Moving on: Reference material, communication	10
Knowing your audience	10
Finding a role	11
General tips	11
Technical writers: Not just a writer	12
Conclusion	13
Bonus: What tools do you use for your writing?	14
Extra bonus: An anecdote	14
A Day in the Life	15
The high-level	15
The days when I write more	15
The days when I write less	16
Is writing customer-facing documentation technical writing?	18
What <i>is</i> technical writing?	18
Types of Documentation	19
Product Documentation	19
Blog Posts	20
Open Source Documentation	21
More Documentation	22
Conclusion	22
Implicitly downplaying knowledge in technical communication	23
Writing introductions in technical tutorials	24
Outlines	26
Bullet Points	26
Headings	26
First Sentences	28

Clarity	30
Challenging assumptions	30
How many words you use to explain a concept	31
Avoid superfluous language	32
Jargon	34
Comparing the excerpts	35
Which example would you send in an email to a friend?	36
Style	37
Growing into style	37
Style outside of professional writing	38
Lists	39
Lists in product documentation	39
Lists in blog posts	40
Balance	41
Placeholders	42
Code Snippets	43
Dependencies	43
Minimum viable code snippets	43
Explanations	44
Substitutions	45
Sharing feedback	46
Effective Examples	47
What makes a good example in technical writing?	47
How I used examples in a blog post	47
Document limitations	49
Example bounds	49
Callout Boxes	51
Warning Callouts	51
Note Callouts	51
Tip Callouts	52
Consistent Examples	54
Run-on Sentences	56

Duplicate Information	57
Pillar content	57
Keeping duplicate information in check	57
Context and repeating information	58
Duplicate information across platforms	58
How-To Outline	59
The how-to outline	59
Walking through the outline	60
Navigation Structure	62
Navigation Structure	63
Navigation Links	66
Technical content and a-ha moments	68
Feature Releases	69
What is the feature?	69
Who should use the feature?	69
For whom is the feature available?	70
How can the reader use the feature?	70
Conclusion	71
Authoring Tools	72
Directing contributors to use specific tools	72
Tool choices	72
Deprecating Content	74
Communicating with the team about deprecations	74
The deprecation lifecycle	75
Deprecating APIs	75
Deprecating content	76
Conclusion	77
Facilitating Ideas	78
Ideating	78
Prioritising	79
Plates: Don't fill them too much	79
Internal Code Documentation Requirements	81
Internal Dry Run	83

Holistic Documentation Reviews	85
Committing to a review	85
What to ask, and how	85
User research for documentation reviews	86
Scoping	87
Making the changes	87
Announcing the project, and review	88
Publishing Contributor Blog Posts	90
Publishing an article: (Maybe) More than you think	90
Authorship	91
Publishing the post	91
Reviewing the Wolfram Language Documentation	93
Reviewing Digital Ocean’s Documentation	97
Setting the reader on the right path: A version selector	97
The introduction	98
Explaining the “whys”	98
Copyable code snippets	99
Highlighted variables to substitute	99
Multi-step code examples are well written	100
Helping readers build a solid foundation of knowledge	101
Referring to next steps	101
My conclusion	102
Being a Technical Writer	103
Exercise: Review a Technical Article	105
What is Image Classification?	106
Image Classification Use Cases	108
How to Classify Images with Roboflow	109
Image Classification Compared to Other Algorithms	113
Conclusion	114
ISBN	115

License

Software Technical Writing: A Guidebook by James Gallagher is licensed under Attribution-NonCommercial 4.0 International. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/4.0/>.

Contact: readers@jamesg.blog

Introduction

You may wonder: how can I write technical content? Do I need to be a great coder? Do I need to have a background in writing? Let me answer those last two questions now: no. Writing is all about communication, as I discuss throughout this book. If you have some technical skills and enjoy refining your communication skills, you have the mindset you need to write technical content. That first question, “how can I write technical content?”, is the subject of this book.

In December 2023, I decided to write a series called *Advent of Technical Writers*, in which I wrote 24 blog posts on technical writing. Written without any particular order in mind, I put my fingers to my keyboard with one guiding question in mind: what knowledge informs how I work the most? I wanted to document the practices that I use to write, all of which have been learned from reading, experimentation, and receiving feedback from thoughtful, detailed editors. Further “bonus” posts were published in the days following the official conclusion of the series.

The series was published on my blog, *James’ Coffee Blog*, receiving attention across various communities. I covered everything from a day in the life of a technical writer all the way to how to make your writing clear and effective. Along the way, I shared both the mindset I have when working on technical content as well as examples to reinforce the points I make. Toward the end of the series, a thought came to mind: should I publish this as a book? In your hands – or on your screens, depending on your reading medium of choice – right now is the result from the “yes!” I proclaimed as I set out to produce this book.

In this document, we will cover essential and practical knowledge that you can use to produce technical content.

This book is positioned toward people who want to start writing technical documents, or who are early in their careers and are looking to refine their skills.

This book will be focused on what I have learned from writing software documentation, code documentation, and technical tutorials. The world of technical writing is vast, encompassing everything from writing instruction manuals for hardware to producing online reference material for programming languages. The words herein may not be as directly applicable to some fields of technical writing as my core areas of expertise, but much of what I cover has broad relevance.

We are going to cover topics such as:

- What a technical writer does;
- How to write clear content;
- How to structure knowledge bases of documentation;
- Working with an organization to improve documentation, and more.

I hope you find the materials in this book a useful guide in your start to technical writing. There is no expectation to read the book in one sitting. You may find it helpful to read this book over the space of weeks or months, incorporating what you learn into your writing as you go. Indeed, there is no better way to learn writing than to practice.

My experience starting as a technical writer

In this post, I am going to reflect on some of my experience as a technical writer, in the hopes that I can impart some information that is useful to you. This is not a “how to become a technical writer” post, or anything of that nature; there are plenty of other places you can go for that. Instead, I want to speak about technical writing from my viewpoint; from what I have seen, heard, and worked on. I will talk exclusively about software technical writing since that is what I know.

If you want to read some of my work before reading this post, or want some as a reference, here are a few samples:

- What is Zero-Shot Classification?¹ (Explainer)
- DINO-GPT4-V: Use GPT-4V in a Two-Stage Detection Model² (Technical tutorial)
- Launch: Advanced Dataset Search Filters, Operators, and Logic³ (Product announcement)
- How to Use FastViT⁴ (Technical tutorial)

Why technical writing?

Think back to when you were last stuck on a coding problem. Really stuck. I will guess that you went to a site like Google, Stack Overflow, or the documentation for the language or package you were using to search for an answer? You might have read a few different pieces of material, then found an answer. Your code works; you can move on to the next challenge.

Technical writers help create that content.

Have you ever encountered a library that you wanted to use but the documentation isn't great? Or the documentation is missing information you need to be productive? Technical writers help ensure that doesn't happen.

As a technical writer, you can help people learn new skills. You get to decide how someone learns a skill. What they should learn first, what path they should take. This is true whether you are writing a tutorial (i.e. how to calculate an image embedding), a piece of reference material (i.e. how to use a library function), or anything else. You get to use your experience to help others have a smooth learning experience.

My motivation as a technical writer stems from being able to help others learn. If I had to explain what I do to someone who doesn't code, I would say one of two things:

¹<https://blog.roboflow.com/what-is-zero-shot-classification/>

²<https://blog.roboflow.com/dino-gpt-4v/>

³<https://blog.roboflow.com/advanced-dataset-search/>

⁴<https://blog.roboflow.com/how-to-use-fastvit/>

1. I write documentation for software companies (this sounds a tad boring, but it is accurate and sometimes this is what I need and I like this description), or;
2. I help people teach computers how to see (I document computer vision software professionally).

That second point, something I came up with earlier this year, brings me a lot of motivation. I build some computer vision technology, but I spend most of my time helping people use what my team has made. I love doing this, and my team enjoys having someone who can take ownership over documentation.

As a technical writer, you will learn a lot. Every day.

Before I started my current job, in which I document computer vision, I didn't know much about computer vision. Now, I can confidently explain what CLIP is and how you can use it to solve various vision problems. I can explain how to identify the colours in a segmentation mask. I can explain how to efficiently filter predictions from state-of-the-art models. I learned almost all of this through my work, with help from my colleagues, a lot of reading, and experimentation.

Mediums of writing

What you do as a technical writer varies depending on for whom you work and what work you are given. Back at the start of my career, I wrote technical tutorials. I had done plenty of writing in my free time, too, although nothing of particular significance. I was learning. I loved words and language. I enjoyed the process of thinking through how to explain a concept.

My first job was to write about Python. I wrote tutorials for beginners, the kind of guide one might read as they are learning the fundamentals of the language. I wrote about what I knew. Nothing was particularly groundbreaking, but I loved the work. I got to think about a topic and explore how I wanted to communicate it. Indeed, a technical writer must be an effective communicator. A communicator who can code well.

When I started to write about technology, I noticed something: a particular interest in documenting how people can use technology, and how I use technology. I wrote more in my spare time, exploring different mediums of writing. I had, up until this point, learned about two different types of technical writing:

- Tutorials, in which I walk through how to accomplish a specific task, and;
- Retrospective blog posts, in which I walk through how I did something, without necessarily showing exactly how I did it.

The former was professional and the latter was for fun. With every post I wrote, and write, I gained more confidence. I refined my format. My style. The latter became part

of how I made side projects. I would write a software project then write about how I made it. The inspiration for this was largely the IndieWeb community, in which members are encouraged to document their projects. Writing about how I did something could start a discussion. In both the aforementioned formats, I could teach someone something new.

In my spare time, I also wrote reference material. For example, I worked on the IndieWeb Utils Python library. This library features many utilities that are useful for building social web applications. In this project, I learned more about software documentation: setting up a documentation tool (in Python), code documentation standards, working with someone else on code documentation, and more. I learn a lot from projects in my spare time.

I have written numerous explainers, too. Explainers are writing in which you define and explain a concept, in depth, potentially without reference to any code. In my mind, explainers often start with “What is?”. For example, “what is an image embedding?”

Moving on: Reference material, communication

I moved on to a job in which I was documenting software, referencing what I had previously written as examples of my work. My new role, where I work on the Roboflow marketing team, opened up new worlds of technical writing. Whereas I had previously written tutorials for work and reviews of how I made a project in my free time, I was now empowered to spend more time on examples. Digging deep into a particular subject matter, making an example project, and then writing about how someone could make that same example.

Earlier I mentioned that technical writers need to be effective communicators who can code well. You don’t need to be a brilliant coder; your talent can shine through in how you explain concepts. With every tutorial I write, I ask myself “what do I expect someone to be able to do at the end of this guide?” I work backwards in my mind through the steps someone would need to accomplish a goal, then I document those steps. With every piece of reference documentation I write, I ask: will this give someone what they need to use this library?

Knowing your audience

Knowing your audience is essential, a skill you will pick up the more you write and the more you work with others. Professionally, I write guides about computer vision, a field of machine learning that focuses on images. Computer vision is broken down into a few task types, including object detection, classification, and segmentation. Object

detection is a method of finding the general location of objects in images. Classification refers to assigning one or more labels from a limited set of categories to an image. Segmentation refers to finding, at the pixel level, the location of objects in images.

Sometimes I write beginner material, like “what is object detection?” In such posts, I know I have to restrict my vocabulary. In all cases, I need to make sure any term I use that someone might not know is defined in my work, but in beginner posts there are many words I know I can’t use. In an article like “what is object detection,” I want someone to be able to answer that question in their own words by the end of the post; the more jargon I use, the harder that is.

Sometimes I write more advanced material, covering a topic like how to use a new machine learning model. In such a piece, I introduce the topic (the model) and explain, in detail and step-by-step, how someone can use the model. I can assume familiarity with machine learning, so I don’t have to define common terms like object detection or classification. But I can’t assume familiarity with the model, since someone searching “how to” use a model likely has limited knowledge of that model. It is my job to give people the knowledge they need.

Finding a role

I have found technology startups to be an excellent place to look for technical writing roles. These roles often pair with other responsibilities and fall under a marketing team. For example, you might be a “technical marketer” (my job title), with technical writing as one of the ways in which you are going to grow a project. I have worked for two early stage startups as a technical writer. In both roles, I have had a lot of ownership over what I do. In my current role, I have been given excellent support and feedback to help me grow, but this is not always a given. Startups have limited resources; starting out may be uncomfortable unless you have already done some writing in your free time (which is a great idea!). With that said, I have a bias: this is the path I took.

I don’t know a lot about larger companies, but I do know that there is always a need for someone who can document software. From the smallest Python package to the largest database solution in the world, documentation is essential.

General tips

Write, a lot. Expect that some of what you write will not be good. I have written many a blog post or tutorial and thought “I don’t like this too much.” As you write more, you will likely become more “picky”. Or, in other words, you are developing a taste. You are learning what you like and what you don’t like in writing. A few years in, I can now

read a sentence and start picking it apart; this skill is useful when I edit others' work, and do other tasks that aren't necessarily technical writing like copywriting.

Get feedback from a good editor. I often skim through the editorial feedback, having trust in my editor. Some comments from your editor will define your work. Here are some examples from my work of my editor giving a comment that I think about regularly when I produce new content:

- Make one amazing example that you explain throughout your work. Referencing lots of “for example” cases can be confusing.
- Make sure abbreviations are in their full form when you first use them (i.e. “Retrieval Augmented Generation (RAG) is ...”, then you can say RAG later on).
- Keep paragraphs to a few sentences.
- Explain exactly what a long code snippet does, step-by-step. This is personal preference, but something I think is valuable. If someone is not comfortable with the code, the text explanation will help them reinforce their knowledge.
- Use simple words where possible. “use” instead of “utilize”.
- Be specific about what a given piece of content is going to help someone do. I like to end conclusions with “In this guide, we will...” and list 2-3 things that someone can expect to have done.
- Add example outputs from code. Add visuals where appropriate.

If you don't have a good editor available, experiment with new styles and continue to push yourself. If you are writing a personal blog post, ask a friend for feedback. When I started writing, I wasn't too comfortable with active feedback. The right editor helped me open up through constructive and detailed feedback that never made me feel like I had made a mistake. Building a growth mindset (“how can I make this better?” vs. “what did I do wrong?”) has taken a lot of work; it is something I refine every week through the course of my job.

Technical writers: Not just a writer

I mentioned earlier that I am a “technical marketer”, which encompasses more than writing. I maintain open source software, contribute to our product, and more. With that said, even if my job was “technical writer” I would still not be writing and editing my work all the time. Being a technical writer involves not just writing, but:

1. Iterating on your guides over time when you receive feedback.
2. Helping team members find your content. For instance, I help our sales and customer success teams find materials that are relevant for prospects and customers, allowing them to be more effective.

3. Assisting with the architecture of documentation. This refers to how pages connect, table of contents structure in your content, and the general flow of pages.
4. Creating documentation infrastructure. This refers to creating mini documentation sites using tools like `mkdocs`, creating automated actions for generating documentation on GitHub, and all the other little tasks that keep documentation running.
5. Writing tips for your colleagues to help them learn how documentation works at your organization.
6. Coordinate with designers to create visual assets, giving them the explanations they need to do their best work. (Designers are magical – thank you for all that you do!).

At my work, I actively encourage people from all parts of the business to contribute to our content, whether their contributions are technical writing, product explainers, or anything else that would help a customer or a member of our audience. I help brainstorm ideas, provide feedback, edit content, and more. I love helping people explore writing. Writing a blog post gives a sense of accomplishment. Helping someone get to that feeling is humbling, especially if they don't write too often.

Conclusion

Technical writers are often behind the scenes, working away to help people use tools effectively. As a technical writer, you are tasked with knowing something inside out and distilling that knowledge into material that will help someone learn something new, create something, or solve a problem. With every paragraph, code snippet, and example, you can help someone feel more comfortable with a topic. You can educate, empower.

You do not have to be the best coder in the world. Being able to communicate what you have learned in a way that others understand is your job.

My job is roughly a 30/40/30 balance between coding, writing, and other tasks such as meetings, reviewing content, giving feedback on technical initiatives, and more. With that said, as aforementioned, my job scope is broader than technical writing. This balance works well for me: I don't enjoy coding all day unless I am working on something that really excites me, and even then I cannot sustain that motivation for long periods of time. With technical writing, I can code, document, work with a team, and feel like I can help people.

Bonus: What tools do you use for your writing?

The inspiration for this article was a blog post published on OpenSource.net⁵. The initial structure of this post was going to be answering, one-by-one, the questions posited in their post. I decided this would be too constraining. I think I covered most of the questions above, though, with one exception: what tools I use?

I use Google Docs for professional writing, Typora for personal writing (and professional writing if I have an unreliable internet connection), and Visual Studio Code for coding. I don't use Grammarly, etc. Over the years I have come to learn what I do and don't like. Every day I learn more about how words interact; software that suggests how words should be structured is not empowering or exciting. I rely on Google Docs to identify typos.

I write best when I have long, uninterrupted periods of time during which to write. I am now accustomed to the occasional piece of writing that needs to be done on a short deadline. This is a muscle I need to build more.

Extra bonus: An anecdote

The term “technical writer” is vague. I met an older man in a San Francisco diner earlier this year. We got chatting; he is a local to the diner. We spoke about everything from travels to our jobs. I noted I was a technical writer, to which he responded that I “write the manuals.” The man worked in software. This interpretation has stuck in my head as a humbling reminder of what I do. His words made me feel proud at what I do. I write the manuals, or as I have come to say now, the documentation.

⁵<https://opensource.net/get-started-with-technical-writing/>

A Day in the Life

The high-level

My job is Technical Marketer for a computer vision company. On the average day, I am striving toward helping to increase adoption for our products, which range from hosted solutions to open source utilities. I work with teams on product announcements, write technical tutorials that show how to use our products or solve specific computer vision problems, coordinate open source documentation across different projects, and more.

There are two categories of days in my work: those in which I spend most of my time writing, and those that are spread out across all of the other technical marketing tasks that I work on.

The days when I write more

Every week, I align on the topics that I should write about with my manager. This may be blog posts, documentation pages, product announcements, or everything in between. This conversation usually happens on a Friday. We go back and forth about the key initiatives that are being worked on in the company and industry trends to inform what is a priority for the next week. These discussions are crucial for both ideation and alignment. By the end, I usually have a few topics to write about and know which are more or less important given our priorities.

I start writing when I feel most equipped to write a piece of documentation. For new products that need to be documented, I often have to ask questions. I write these down, then ask them to the relevant parties. I take notes. I ask more questions. It is important for me to know as much about a product as possible. How we want to position it. The usage we want to encourage. How much the product is going to cost (if anything). For whom the product will be available. How the product integrates with other products.

When I am writing a blog post that requires code, I work on an example. If I encounter issues while working with an API, I share the feedback with the team. Again, I may ask questions to other developers about the code I have written and an API. The more information I have, the more equipped I feel to write about a topic.

Then, I get to work writing. I usually prepare outlines in the afternoons which I will then turn into full posts in the morning. These outlines usually start as a few bullet points or headings that I have written out. I will refer to any information that is documented in messages and the product itself as I work. During these times, I can be heads-down

for hours writing different pieces of content. This is the time in which I feel the greatest state of “flow” per se. I am focused on writing.

The job of a technical writer involves going back and forth between documents, products, examples, and code to produce content. Thus, when I say “writing”, I mean both writing and doing all of the reference work I need to prepare a post.

In a day, I might write two or three blog posts, or a few pages of open source documentation.

The days when I write less

A technical writer doesn’t write all day, every day. My role involves other responsibilities that are related to marketing but are not explicitly writing. Today was one of the days when I write less.

I started today by working on a documentation project. This project is going to span multiple days. I reviewed some work I did at the end of last week to get started, reviewed code snippets, ensured all code had the right `import` statements (a potential “gotcha” in the content I am writing right now), and created visualizations.

Visualizations are important but they take a fair bit of time to create. I usually create visualizations after writing some content. By “create a visualization,” this meant using an open source software package I co-maintain at work to show computer vision model predictions in different ways. I sometimes create visualizations and think “this could be better.” Today was one of those days. I created multiple different examples, striving toward the one that best illustrated what the functions I was defining do.

Later in the day, I moved on to more administrative tasks. I invited team members to our Google Search Console accounts for technical documentation that I just set up. I filed a few PRs to add code search and custom open graph images to our documentation sites, features available in the `mkd docs` software on which we depend for Python documentation. I went back and forth on one of these PRs because there was a dependency issue we had to resolve.

Then, I had content to edit. Release notes came in for a project that I reviewed and polished. I used this as an opportunity to ask questions about the project, knowing that I will have more documentation to write about the feature when it has been released. After editing those release notes, I edited a README for a new section in an open source project. I provided feedback and suggestions, with some additional writing tips for the edification of the person whose work I was reviewing.

I coordinated other pieces of content, too. A colleague reached out to ask if we could meet to discuss writing a blog post. When we meet, I will do what I can to help turn the

idea into a reality. I provided some notes on an existing piece of content, deferring to our marketing lead to do the final review because it was getting toward the end of the day and we wanted to publish the content today.

And then there were all the other little tasks that I had timeboxed for today that I don't recall.

While today was one of the days I wrote less, I have a documentation site to finish, a blog post to write, and significant revisions to make to a piece of documentation that is now outdated. I love writing, but the balance between the days when I write a lot and the days when I don't write too much is helpful. Writing all the time can get tiring.

I love the part of my role that involves helping others write content. It is not only my job to write content, but to help others do the same. Empowering my team members to write – brainstorming ideas, making them feel comfortable with the process, providing feedback – is one of the most fulfilling parts of my job. I am excited to help get some posts across the finish line over the next week written by colleagues who are contributing to our blog for the first time.

Is writing customer-facing documentation technical writing?

Is writing customer-facing documentation technical writing?

I wanted to share more thoughts on this question. What better place to do so than my blog?

TL;DR: Yes.

But why? Let's talk about that.

What *is* technical writing?

“Technical writing” is a broad term. Technical writing can mean writing:

1. Manuals on how to use a physical product
2. SDK instructions for software developers
3. Instructions that show how to use a software product
4. Internal reference material for engineers in an organization
5. Educational tutorials
6. Among many other things

The type of technical writing you will do will depend on the organization for whom you work.

I work for a software company that develops computer vision technology. Most of my work involves documenting how to use our products. Some of the products I document are customer-facing, such as Roboflow Annotate, a tool that lets you annotate images for use in training computer vision models. Other products are documented for use by developers, such as our hosted API.

I prefer to think about “technical writing” less not as a specific task but as your approach to writing.

If your job involves (or would involve) writing documentation that explains products, software, hardware, or internal tools, you are a technical writer. If most of your writing is internal documents that aren't for customers or learners (i.e. PRDs, which document product requirements), you are practicing the skill of technical writing in a way. But, you probably have another core job function.

Types of Documentation

It is rarely the case that your software documentation is in a single place. It is important to know different types of documentation so that you know what to write and where it should go. Furthermore, how you write content – a blog post, a product documentation page, open source documentation – will differ depending on where the documentation is going to be published.

Let's talk through a few types of documentation you may encounter in software companies. We will talk about:

- Product documentation
- Blog posts
- Open source documentation

Product Documentation

Product documentation collates all the information someone needs to use a product. Depending on your product, this documentation is include more images than other types.

The primary goal of product documentation is to educate the reader on *how* to use your product. Through each page, a reader should be able to accomplish a specific task. For example, Roboflow makes computer vision technology. Our documentation covers everything from creating a Project (a feature in our product) to deploying vision models with our technology.

Product documentation not only shows someone *how* to do something, but it may also introduce someone to new features in your platform. For example, someone may come to the Roboflow documentation to learn more about how to annotate an image. They may then explore the Annotate section of our documentation to find new features.

You should document as much of the common actions that users can take on your platform as possible, with visual aids to assist readers in their understanding.

Here are a few examples of product documentation:

- Supabase⁶
- Elasticsearch⁷
- Roboflow⁸

⁶<https://supabase.com/docs>

⁷<https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>

⁸<https://docs.roboflow.com>

Blog Posts

“Blog posts”, like “technical writing”, is a broad term. A blog post can be:

- A post that announces a new product and shows how to use it. This should not replace writing a page in your product documentation. Your product documentation should be the comprehensive, authoritative source on all products and how to use them.
- A tutorial on how to use a product for a particular use case. The “for a particular use case” is important here. Whereas product documentation should be as general as possible, you can use blog posts to show how a particular segment of users can use your product. For example, Roboflow has blog posts on how to use our video inference API to build a search engine. We document how to use the video inference API in our product documentation. The video search blog post explores a particular use case in more depth.
- A general education post. For example, Roboflow has posts that define terms like object detection, zero-shot classification, keypoint detection, and more. These posts are intended to help people learn about a concept, with reference to our product where relevant. The foundation knowledge built in such a post can then be used to accomplish a task, such as may be documented in a tutorial.

There are many other types of blog post that are not related to technical writing. Thus, I will not talk about them here.

Blog posts are an effective way to build knowledge on best practices and to create a repository of general information about jargon and topics that users of your product may encounter.

For example, you can use the Roboflow blog to learn about how to use the Segment Anything model. We have Segment Anything support for automated labelling in our product, which is documented in our product documentation. We also have a blog post that shows how to experiment with the model yourself, which is catered to an audience with more machine learning experience.

Examples of blog posts include:

- How to Deploy Computer Vision Models Offline⁹
- Launch: Roboflow Video Inference API¹⁰
- What is Zero-Shot Classification?¹¹

⁹<https://blog.roboflow.com/deploy-computer-vision-models-offline/>

¹⁰<https://blog.roboflow.com/roboflow-video-inference-api/>

¹¹<https://blog.roboflow.com/what-is-zero-shot-classification/>

Open Source Documentation

Open source documentation is like product documentation in that your goal should be to give someone all the information they need to use your product. With that said, writing open source documentation differs from writing product documentation in many ways.

Open source documentation involves several aspects, including writing:

1. A README, which shows someone how to install and use your software;
2. Contributing guidelines and a code of conduct;
3. Guides which show how to use key parts of your software;
4. Code documentation.

A small project may only have a README, and ideally contributing guidelines and a code of conduct. Larger projects with more features tend to have dedicated documentation sites which explain each feature in depth. One such example is Autodistill, an open source Python package that you can use to auto-label images.

Autodistill has documentation which covers:

1. Getting started with auto-labeling data using Autodistill.
2. How to use key library features like `plot()` to plot an image with labels, `compare()` to compare two or more models for auto-labelling, among others.
3. Information about the Autodistill ecosystem of packages, including what packages are available and how to use them.

Some of this documentation, such as the getting started guide, is written on a documentation website by hand. Some documentation is auto-generated from the code docstrings. We use `mkdocs` to generate and render documentation.

The amount of documentation you need will vary. At a minimum, you should have all the documentation someone needs to use your software with minimal friction. Generally, the more documentation you have (that you are willing and able to maintain!), the better.

Examples of open source documentation:

- supervision¹²
- Hugging Face Transformers¹³
- llamafile¹⁴

¹²<https://supervision.roboflow.com>

¹³<https://huggingface.co/docs/transformers/index>

¹⁴<https://github.com/Mozilla-Ocho/llamafile/blob/main/README.md>

More Documentation

There are many more forms of documentation you will encounter, but I have had less or no experience writing them so the extent to which I can lend insights is minimal. With that said, I can share some high-level information. Here are a few other forms of documentation that exist in software:

1. **Standards:** These are well-researched documents that take extensive periods of time to make. Standards typically go through a robust technical review process (i.e. the W3C Candidate Recommendation process for the W3C) to ensure the technical details of the standard.
2. **Manuals:** Manuals document how to use a product in full. You could think of online product documentation as a manual, except spread over many pages.
3. **Help Center Documentation:** This may be used in addition to product documentation to answer specific questions such as how to delete an account. Help centers often provide both documentation and a form through which users can request support. Help center documentation can become duplicative, however. Be sure that your help center does not aim to document the key features in your product that are in your main product documentation.

Conclusion

The type of documentation you write as a technical writer or engineer will depend on your role. For example, I write product documentation, blog posts that showcase product features for specific use cases, and open source documentation. That is because my role spans across both growth, marketing, and open source. At a large company, you may focus solely on product or code documentation, depending on your role.

Implicitly downplaying knowledge in technical communication

I recommend avoiding the words “just, simply, easy” and their variants in technical writing, particularly in print.

These words are usually redundant and add unnecessary assumptions to your content. Seeing words like “just” or “simply” are discouraging, particularly to people who have limited knowledge of the subject matter.

Consider the following two sentences, both pertaining to the target audience for the IndieWeb community:

You just need to have your own website or be interested in setting one up.

To join, you should be interested in personal websites. You do not need to have a site!

In the first example, I use “just”. This is inappropriate because “just” setting up a website or having one is a big step. Setting up a website is non-trivial.

In the second text, I convey the same information — an interest in or having a personal website is a requirement for joining the community — but the second sentence is more encouraging and less presumptuous.

The first sentence was from my writing two years ago. Since thinking more about this subject, I have become more proactive in catching when I use redundant words. When I notice I use a word like “just” or “easy” (in context such as “Getting started is easy.”), I remove them.

Let’s walk through one more example:

I asked for a specific format, so as to ensure the prompt was easy to parse.

I asked for a specific format, to aid in parsing the prompt.

Again, I’m conveying the same information; the delivery is different.

Avoiding words like “easy” and “just” are an essential part of technical editing and communicating. Our goal is to empower people with knowledge, and doing so necessitates our recognizing the amount of work it has taken ourselves to learn what we know.

Writing introductions in technical tutorials

1. Introduce the main concept being discussed in an article, with relevant background.
2. Reference an example to which I will refer later in the post (optional, depending on the length of the background in the opening paragraph(s)).
3. Summarise, in a few points, what will be discussed in the article.
4. Show an example of what someone will accomplish by the end of the article, if possible (most relevant for visual guides such as those that pertain to web design, image editing, computer vision, etc.).
5. Conclude with a welcoming sentence that eases the reader into the main content.

Here is an example from a recent tutorial I wrote for work on SAHI¹⁵, a technique that enables you to detect smaller objects with a greater degree of accuracy using computer vision models:

Detecting small objects is a challenging task in computer vision, yet significant for many use cases. Slicing Aided Hyper Inference^a (SAHI) is a common method of improving the detection accuracy of small objects, which involves running inference over portions of an image then accumulating the results.

In this guide, we are going to show how to detect small objects with SAHI and supervision^b. supervision is a Python package with general utilities for use in computer vision projects. The supervision SAHI implementation is model-agnostic, so you can use it across a range of models. We'll walk through an example of detecting objects of interest in a beach image with SAHI.

Here is an example of an image on which inference was run with and without SAHI:

[image]

Without further ado, let's get started!

^a<https://github.com/obss/sahi?ref=blog.roboflow.com>

^b<https://github.com/roboflow/supervision?ref=blog.roboflow.com>

This example follows the structure I outlined at the beginning of this post. I introduce the technique (SAHI), what we are going to do in the article (detect small objects with SAHI), how we are going to do it (use the supervision Python package), some important information about SAHI (it is model-agnostic), an example we are going to talk through (a beach image), and include a photo showing the results of the guide. You can see the photo yourself in the SAHI blog post¹⁶.

¹⁵<https://blog.roboflow.com/how-to-use-sahi-to-detect-small-objects/>

¹⁶<https://blog.roboflow.com/how-to-use-sahi-to-detect-small-objects/>

I want my introductions to set expectations, ease a reader into a topic, show what we are going to do, and explain how we'll do it.

I like to end posts with a variant of “Without further ado, let's get started!” or “Let's begin!” because I think it sets a nice tone for a post. I want readers to feel like they are following a guide *with* me, rather than being told how something is. This is important to me. The key to good technical writing is that it is tailored to an audience and approachable for that audience.

This is my take on an introduction. Everyone has their own style, informed by their experience and the types of content they write; there are few hard rules about introductions. Ultimately, it is up to you, the writer, to figure out what you want to say and how you want to say it!

Outlines

1. A few bullet points that note what I want to talk about, or;
2. A few headings that summarise the content I want to include in an article, in order of how they should appear in the guide.

Bullet Points

Writing a few bullet points in the document you are going to write in (i.e. a Google Doc, or whatever your editor of choice) helps you avoid having a blank page at the beginning. You have text, in front of your eyes, that can ground you. You can write down whatever points you want to make, in as much or as little detail as you want. These points can help you get started. When you start writing, you might find that your bullet points are easier to expand than if you were writing only from the knowledge in your head.

I sometimes start blog posts with a few bullet points. I do this a lot when I am not yet ready to write a post but I have a few ideas that I want to remember. Here are the bullet points I wrote for this guide as an example:

- Bullet points, or a document structure
- Helps with thinking about what you want to say and in what order
- You can always change later

These bullet points are rough and may be difficult to interpret since you didn't write them. Notes always make more sense to the author. Above, all bullet points are concise and express an idea that I wanted to mention. I didn't go into too much detail, but you could do so in your bullet points. Notice how that first bullet point was fleshed out into the first few paragraphs of this post. I will explore the themes in the other two bullet points later in this post

With the above bullet points, I validated that I had topics to talk about, which helped me build more confidence. I also knew that I wasn't starting from a blank slate. This is a big deal. Starting from a blank page can be intimidating.

Headings

I also find writing example headings useful. I write the headings in the order they will appear in a document. For example, here are the headings that I wrote for my guide on deploying vision models offline¹⁷:

¹⁷<https://blog.roboflow.com/deploy-computer-vision-models-offline/>

1. What is Roboflow Inference?
2. Preparation: Upload or Train a Model on Roboflow
3. Step #1: Set Up Roboflow Inference
4. Step #2: Run a Vision Model on an Image
5. Step #3: Run a Vision Model on a Webcam
6. Conclusion

(I don't have the raw headings I wrote, so these are from the article itself. My raw headings were not too far from this.)

By writing these headings, I already had text on the page. I was also able to build a better picture about what I wanted to talk about and in what order. I knew I needed to include:

1. An introduction to the solution (Roboflow Inference).
2. Preparatory information, which in this case was uploading or training a model to our platform.
3. How to set up Inference.
4. How to use Inference.

With these headings ready, I could get to work, knowing I already had something of an outline on the page. As I wrote, I could spend less time thinking about what section I was going to write next, and more time on expanding each title into a full section. Often, my titles change. I realise I need to add more sections, or remove some. That is okay. What matters is getting started on a piece; this technique has helped me do so.

First Sentences

In many cases, I start the main content in documentation with 1-2 sentences that introduce a feature and what we will accomplish. Here is an example:

With Supervision, you can easily annotate^a predictions obtained from a variety of object detection and segmentation models. This document outlines how to run inference using the Ultralytics^b YOLOv8 model, load these predictions into Supervision, and annotate the image.

^a<https://supervision.roboflow.com/annotators/>

^b<https://github.com/ultralytics/ultralytics>

This text appears at the start of a guide on how to detect and annotate objects¹⁸ with the `supervision` Python package. I start the article by introducing the topic: you can annotate predictions from different models. Then, I talk about what we are going to accomplish. After these sentences, the article begins.

Here is an example of the first sentences in a guide called “Run a Fine-Tuned Model”:

With Inference, you can run any of the 50,000+ models available on Roboflow Universe. You can also run private, fine-tuned models that you have trained or uploaded to Roboflow.

This guide proceeds to document running models from Universe and fine-tuned models.

The language above was crafted to be clear and concise. With these two sentences, the reader learns:

1. Roboflow Universe has 50,000 models they can run.
2. You can deploy those models on Inference (“Inference” is the product name).
3. You can run fine-tuned models on Inference, too.


Then the guide explains how.

Alternatively, you may opt to go directly in to the post content, such as in the example below, taken from a documentation page shows how to add team members to an account¹⁹:

To invite a team member to a Workspace, add their email address in the Workspace Members section of the workspaces settings.

¹⁸https://supervision.roboflow.com/how_to/detect_and_annotate/

¹⁹<https://docs.roboflow.com/workspaces/adding-team-members-to-a-workspace>

 [image showing how to do this]

Introducing that you can add team members is not relevant, since it can be assumed from the title of the documentation that you can add team members. Thus, the page delves immediately into the intent of the page, without preamble.

Clarity

Challenging assumptions

Technical writers need to make assumptions. You need to assume some level of familiarity with a concept for each piece of content you write and use that assumption to inform what you say and how you say it. With that said, you should actively ask the question “is this assumption correct?”

Here is part of an introduction from a guide I wrote on how to analyze the color of a product²⁰:

In this guide, we are going to discuss how to analyze product color using computer vision. We will walk through an example of taking an image of a t-shirt and identifying the color of the shirt.

To analyze the product color in an image, we will:

1. Use a segmentation model to identify the precise location of the object whose color we want to check, and;
2. Use clustering to identify the most common color(s) in the image.

The title of the guide is “How to Analyze Product Color in Quality Assurance Processes”. This guide was written for anyone who wants to analyze product colors. I cannot assume that someone knows that computer vision is the solution.

I explicitly say that we are going to “analyze product color using computer vision.” I walk through the exact steps. When I say “use a segmentation model,” I imply what that means (“identify the precise location of the object whose color we want to check”). In hindsight, I could make this even clearer by saying “computer vision system,” then introducing the concept of “segmentation model” later. I say that we are going to use clustering to identify colors in an image. I say the how and the what.

The two bullet points above could be expressed as:

To analyze the product color in an image, we will:

1. Use a segmentation model to find objects of interest;
2. Use k-means clustering to identify common colours, then select the top color.

Compare the text above to the previous text. I say “segmentation model” without saying exactly what that entails. If a reader knows a bit about computer vision, they may wonder “why segmentation?” Therein lies a question I have sparked. A reader might not know what k-means is, even though that is the solution.

²⁰<https://blog.roboflow.com/how-to-analyze-product-color/>

When you use jargon, make sure it is well explained for your audience. Don't assume that someone knows something even if you do.

This, like so many aspects of writing, is a muscle. Something can *feel so obvious* to you that it doesn't need to be said. A way to get better at this is to always work with an editor. An editor will point things out in your writing that you never notice.

One pattern I like to use is:

1. Define
2. Explain

Another way to think about this is with the following format:

X is Y. This means...

This pattern has helped me get used to stating a concept then explaining it immediately after. Using a new sentence to explain a concept in more depth is helpful. Long sentences can make writing hard to read.

How many words you use to explain a concept

When you are writing, say what you want to say in as few words as you can. Consider this text, which describes how to automatically label images for use in training a vinyl record detection model. This text is technical, so you may want to read the introduction of the quoted post²¹ to learn more context.

First, our code will use Grounding DINO and cut out all vinyl record covers. Then, CLIP will classify each cover and assign a label.

We need to calculate CLIP embeddings for one image that contains each class (SKU) we want to use in classification. Embeddings are numeric representations of an image that encode semantic information about the image. This information can be used to classify objects into categories.

The reference images we use should primarily feature the SKU you want to detect. Featuring multiple products will confuse the model and result in sub-par results.

The text above communicates a significant amount of information, including:

1. The role “Grounding DINO” plays in a task (cut out vinyl record covers).
2. How we cut out vinyl record covers (using “Grounding DINO”).
3. The role “CLIP” plays in the task.
4. CLIP embeddings are necessary to use cLIP.

²¹<https://blog.roboflow.com/label-product-skus/>

5. What embeddings are.
6. What role embeddings play.
7. Among other pieces of information.

With that said, we only used a few words. We say what things are and explain how they connect. It takes surprisingly few words to do this well. Verbosity is not rewarded in technical writing.

The UK government does a great job at communicating with few words. There is a clear impetus to invest heavily in this: their words need to be understood by all citizens. Every page uses short sentences that are clearly linked.

Here are a few examples:

- Help with your energy bills²²
- Educating your child at home²³

I recommend reading over others' writing and asking yourself the question "could this be expressed in fewer words?" This will help you build an editorial eye. Over time, you will start thinking to yourself "oh, this could be simpler." Or "I can use fewer words to say this." It is often easier to analyze others' writing, especially as you begin, to look for patterns. You will learn what you like and don't like.

Avoid superfluous language

One habit I actively fight against, and have noticed in other people who communicate technical concepts in written language, is the tendency to include superfluous words. These are words and phrases that don't add much value to a sentence.

Consider these two statements:

The Roboflow Video Inference API is a hosted solution that enables you to run any Roboflow Inference^a model for video processing. You can run custom object detection, segmentation, and classification models for identifying visual data specific to your domain.

^a<https://inference.roboflow.com/?ref=blog.roboflow.com>

Versus:

The Roboflow Video Inference API is a hosted solution that enables you to run any Roboflow Inference^a model for video processing. This API is designed to let you run custom object detection, segmentation, and classification models for

²²<https://www.gov.uk/get-help-energy-bills>

²³<https://www.gov.uk/home-education>

identifying visual data specific to your domain.

^a<https://inference.roboflow.com/?ref=blog.roboflow.com>

The first sentences are taken from a guide I wrote that introduces a video inference API. I have adjusted the second sentence in the revised version above.

Notice that the second text says “is designed to let you run”. These words don’t add much value to the sentence. We don’t need to say “designed” because the existence of a dedicated API suggests it was designed for a purpose. “to let you run” doesn’t communicate information. An API should enable someone to do something.

Let’s break out these sentences:

You can run custom object detection, segmentation, and classification models for identifying visual data specific to your domain.

Versus:

This API is designed to let you run custom object detection, segmentation, and classification models for identifying visual data specific to your domain.

Both communicate roughly the same information, but the first is more concise. It relies more on what was said in the last sentence and uses fewer words to explain the same concept. The value proposition is clearer in the first sentence. “You can run” is a call to action. More broadly, “you can” is a powerful set of words. Those words can empower the reader and help them feel like a task is achievable. “This API is designed to let you run”, on the other hand, is less interesting and less engaging.

Another category of superfluous language is words that aren’t jargon that could be simpler. Technical writing involves effort for a reader to comprehend. The more complex words you use, the more effort required to understand a concept. For example, use “use” instead of “utilize.” “Process” instead of “pipeline.”

Being more direct, removing superfluous language, and using simpler words where ones are available are details that make a notable difference in a reader’s ability to understand what you are communicating.

Jargon

The audience for which you are writing, and the knowledge they have, should inform several parts of your writing, including how you define terms. In general, make sure that you only use terms that are:

1. Relevant to the documentation or blog post you are writing.
2. Defined in their first use, if they are jargon terms that a reader is unlikely to know.
3. Written in long-form as well as short form if they are abbreviations, in their first instance. Then, used in short form.

Consider the following sentences which define the term “zero-shot classification”, taken from an article by Amazon²⁴:

Zero-shot classification is a paradigm where a model can classify new, unseen examples that belong to classes that were not present in the training data. For example, a language model that has been [sic] trained to understand human language can be used to classify New Year’s resolutions tweets on multiple classes like career, health, and finance, without the language model being explicitly trained on the text classification task. This is in contrast to fine-tuning the model, since the latter implies re-training the model (through transfer learning) while zero-shot learning doesn’t require additional training.

The following diagram illustrates the differences between transfer learning (left) vs. zero-shot learning (right).

[diagram]

Yin et al.^a proposed a framework for creating zero-shot classifiers using natural language inference (NLI). The framework works by posing the sequence to be classified as an NLI premise and constructs a hypothesis from each candidate label. For example, if we want to evaluate whether a sequence belongs to the class `politics`, we could construct a hypothesis of “This text is about politics.” The probabilities for entailment and contradiction are then converted to label probabilities.

...

^a<https://arxiv.org/abs/1909.00161>

Now read this, an excerpt from an article I wrote on zero-shot classification:

Heading: What is Zero-Shot Classification?

Zero-shot classification models are large, pre-trained models that can classify im-

²⁴<https://aws.amazon.com/blogs/machine-learning/zero-shot-text-classification-with-amazon-sagemaker-jumpstart/>

ages without being trained on a particular use case.

One of the most popular zero-shot models is the Contrastive Language-Image Pretraining (CLIP) model developed by OpenAI. Given a list of prompts (i.e. “cat”, “dog”), CLIP can return a similarity score which shows how similar the embedding calculated from each text prompt is to an image embedding. An embedding is a numeric representation of text or an image which can be compared to measure similarity. Embeddings encode semantics, which means that the embedding for “cat” will be closer to an image for a cat than the embedding for “dog”.

You can take the highest embedding similarity as a label for the image.

CLIP was trained on over 400 million pairs of images and text. Through this training process, CLIP developed an understanding of how text relates to images. Thus, you can ask CLIP to classify images by common objects (i.e. “cat”) or by a characteristic of an image (i.e. “park” or “parking lot”). Between these two capabilities lie many possibilities.

Which do you prefer, and why?

(Note: The first example talks about text classification, whereas the second talks about image classification. AWS showed up first in a search for “what is zero shot classification” so I used their example.)

Comparing the excerpts

I want to point out a couple of features of this the second text. First, notice how I defined zero-shot classification in the first sentence. My definition is concise. To make this more effective, I could add “This is in contrast to fine-tuned models, which need to be trained on specific use cases to be used.” I will need to make that update!

Then, I refer to an example, CLIP. I define CLIP, using its long (“Contrastive Language-Image Pretraining”) and short (“CLIP”) forms. I give this topic a few sentences since it is essential to understanding the topic. Then, I talk more about CLIP.

Here, I assume the reader has a cursory understanding of “training”. If someone is reading about zero-shot classification, they probably have some general awareness of machine learning. This is because zero-shot classification is a specialized form of classification. I also assume the reader knows what “classify” means.

We introduce embeddings, which is a technical term. But, they defined, and with reference to an example in context.

In contrast, the first example uses more complex words like “paradigm” which add little value to the definition. The second sentence in the first example is a run on sentence, which makes it difficult to read.

Then, the second example mentions “transfer learning”. Transfer learning is relevant, but it is likely a new machine learning term to the reader. It is explained in a diagram (not pictured above, but present in the original article), but no text is written to supplement and explain the diagram. The article then mentions a phrase like “constructs a hypothesis from each candidate label”, which is relatively hard to understand.

“The probabilities for entailment and contradiction are then converted to label probabilities.” is confusing. Someone with limited knowledge of statistics may wonder if “entailment” and “contradiction” are technical terms with meanings, or gloss over them entirely because they are complex.

From the two excerpts above, there are a few things we can learn. First, concise definitions matter. Second, it is key that you define jargon clearly. Third, you should use language that is intuitive to your audience. Transfer learning, for example, is somewhat difficult to understand. My post doesn’t mention the term because knowing it doesn’t impact your ability to understand and intuit what zero-shot classification is.

Which example would you send in an email to a friend?

If you are used to writing complex technical documents, the first and second examples may seem equally acceptable (and perhaps the second one lacking in detail). Put yourself in the shoes of a friend who is learning about classification and has limited statistics knowledge. Which definition above would you send them in an email to help them build their understanding?

Words matter. Technical jargon is unavoidable, but it is our job as technical writers to help introduce readers to jargon at the right pace. As a technical writer, you should minimise jargon, define jargon when it is used, and avoid assuming knowledge that someone is unlikely to have given the subject matter of an article.

Style

That is style (on more levels than one!).

Technical writing involves a lot of regiment. Many organisations have style guides that document how you should write. Content should be consistent. But, there is still significant room for creativity. We often call this “style.” Style is how you write. The way that you express, explain, structure. The way that you turn an idea from a light bulb above your head (metaphorically), to a finished piece.

Growing into style

Throughout my professional technical writing, there have been varying degrees to which I can apply a style. When I wrote content for beginners who were learning Python, I liked to use coffee shops as examples. To explain data structures, what standard library functions did, and more. I did this because I liked coffee. This was part of my style. I also learned the importance of balance between direct language that fosters understanding and bluntness. I welcome introducing a bit of playfulness into my writing.

In my current role, which involves documenting open source software, writing technical guides, marketing materials, and more, my style comes out in different ways. The way I often end introductions with text like “Without further ado, let’s get started!” or in the way that I include a visual example of what a reader will accomplish by the end of a guide before the end of an introduction.

Don’t worry about finding your own style; you will grow into one as you explore. I have found writing different types of content helpful in building my style. Writing personal content on this blog has been helpful, too. The more types of writing to which you are exposed – both through reading and writing – the more you will get to know how you prefer to communicate.

In writing, there is a place for directness, conciseness, whimsy, visual examples, introducing additional context that you found fascinating while researching a topic, and more. You get to decide how you communicate concepts.

How do you find style? Read, write, get feedback, and keep going. Don’t worry about finding a style. You will grow into one over time. You will find yourself using certain words and phrases over again. You will find that you structure your work in certain ways. The feedback you receive will help inform how you write.

The goal of style is not to be distinct so much as it is to use your unique ability to express concepts in ways that people can understand best.

Style outside of professional writing

Indeed, whimsy and humour, while not appropriate for many professional tasks, can go a long way in personal writing. For example, I just came across a blog post on convolutions²⁵, a mathematical concept with applications in machine learning, that started like this:

Like making engineering students squirm? Have them explain convolution and (if you're barbarous) the convolution theorem. They'll mutter something about sliding windows as they try to escape through one.

Here, the author has taken a more informal tone than most writing I have read on machine learning. The author decided that these words would be effective as a hook to introduce their piece. The author then goes on to say:

Yikes. Let's start *without* calculus: **Convolution is fancy multiplication.**

While “fancy” might not fit in an academic paper, it fits in well in this explainer. When you introduce complex technical topics to a new audience in educational material, informal language can go a long way to make your content more relatable and less dry. The extent to which you use this opportunity? That is style.

²⁵<https://betterexplained.com/articles/intuitive-convolution/>

Lists

I want to talk about lists in the context of ordered or unordered and unordered lists. Ordered lists, which start with numbers, should be used to convey sequential information (i.e. actions to take in a product). Unordered lists, which start with bullet points, can convey any related points

Lists are useful when you have a series of points that you want to convey and highlight. Rather than list the points you want to make with commas – and potentially end up with a paragraph that is difficult to read, depending on how long each point in the list is – you can separate out each point.

I like lists because they:

- Are easy to skim;
- Can improve reading comprehension, and, as a result;
- Give you a tool to highlight key pieces of information

Let's talk through some scenarios where lists can be helpful.

Lists in product documentation

Lists are easy to skim, allowing a reader to easily digest information. This is especially important in product instructions and documentation.

Consider the following introduction to a feature called Health Check²⁶ in Roboflow, a computer vision platform:

Health Check shows a range of statistics about the dataset associated with a project. You can see the following pieces of information:

- Number of images in your dataset;
- Number of annotations;
- Average image size;
- Median image ratio;
- ...

Health check offers nine pieces of information that someone can use to analyze a dataset. Each piece of information is listed in a bullet point (the snippet above is edited for brevity).

A paragraph containing this information would be verbose; a section with a sentence devoted to each would cause the page to balloon in size, without clear value to the

²⁶<https://docs.roboflow.com/datasets/dataset-health-check>

reader. Indeed, the value to the reader is knowing *what* the feature is, how it can help them, and how to access it. The bullet points play a key role in educating the reader *what* the feature (health check) offers. By virtue of the list enumerating features, conveys information needed to evaluate *how* the feature can help them.

You can also use bullet points to list steps, like Supabase does in the “Project setup” section that is embedded across several tutorials on their site (example²⁷):

Let’s create a new Postgres database. This is as simple as starting a new Project in Supabase:

1. Create a new project^a in the Supabase dashboard.
2. Enter your project details. Remember to store your password somewhere safe.

Your database will be available in less than a minute.

Finding your credentials:

You can find your project credentials inside the project settings, including:

- Database credentials: connection strings and connection pooler details.
- API credentials: your serverless API URL and anon / service_role keys.

^a<https://database.new/>

Here, we have two lists. The numbered list enumerate product actions that you should follow. The bullet points highlight two key pieces of information you need: database and API credentials.

Lists in blog posts

I sometimes lists to explain long code snippets. If a post contains a lot of code and text, I find using numbered lists as an effective way to introduce variation in how explanations are presented.

Consider this description, taken from a code snippet on a guide I wrote on using Optical Character Recognition with the Roboflow Inference product²⁸:

In this code, we:

1. Use a fine-tuned model to identify the container number in an image, then;
2. Make a web request to the OCR endpoint to retrieve the text in the image.

²⁷<https://supabase.com/docs/guides/ai/integrations/roboflow>

²⁸<https://blog.roboflow.com/ocr-api/>

You will need to replace two values in the code above:

1. `image.jpg`: The name of the image on which you want to run OCR, and;
2. `API_KEY`: Your Roboflow API key. Learn how to retrieve your Roboflow API key^a.

^a<https://docs.roboflow.com/api-reference/authentication?ref=blog.roboflow.com#retrieve-an-api-key>

I used numbered points twice: first to list what the code does, then to highlight what values the reader needs to replace.

Balance

Indeed, like using a balance of short and long sentences helps make writing more cohesive, using lists throughout a post makes it easier to comprehend content.

With that said, you should use lists conservatively. In this article, I used lists once outside of the example excerpts (which had to include lists because the post this blog post, unlike any other I have written, is about lists!). I find that I use lists no more than a few times in a guide.

You should avoid using lists as the primary method you use to explain concepts. Lists can improve readability, but a long list of bullet points is confusing; finding the information one needs will be difficult.

How do you find balance? Practice. If you write content that contains a lot of lists, consider if some of them could be paragraphs. Some lists only need a bit of revision to be turned into a fully-formed paragraph. Conversely, if you have a dense paragraph that explains a code snippet, it may be worth either dividing into several paragraphs or, if there is not too much information, using ordered or unordered lists.

Placeholders

I like to use placeholder values for images and final code to include in an article when I am writing documentation. The placeholder I use is `[add image]` or `[add code]`. I use `[add image]` when I don't have an image immediately to hand and know I will need one or when the type of visual asset I want to add (i.e. a GIF) isn't supported in my writing tool (Google Docs). I use `[add code]` for the same purpose except for code. I sometimes add placeholders for code when I have a working script to which I am referring but have changes to make (i.e. reduce the number of lines of code, change placeholder values).

When I have written my first draft of the text for an article, I can go back and search for `[add to find values that need to be changed efficiently]`. I can go back and fill in images and code as necessary.

This is a “quick trick” that helps me stay focused on writing. Perhaps it will be helpful to you, too. (But always remember to search for your placeholder. Double check that your work doesn't contain placeholders.)

Code Snippets

Dependencies

First, make sure that you tell the reader exactly what dependencies they need to use code snippets in your guide, and how to install them. This will significantly reduce friction.

Minimum viable code snippets

Second, I recommend striving toward using “minimum viable code snippets.” This is the minimum amount of code I know someone will need to accomplish a particular task.

Consider my guide that explains how to deploy vision models offline²⁹. Before I wrote this guide, I knew that I wanted to teach someone how to deploy vision models offline. I knew this would involve code snippets. I spent some time thinking about what to write, and decided that my code snippets would fall under two themes:

1. Deploying a model offline to run on images, and;
2. Deploying a model offline to run on video.

Then, I wrote code (or copied code from the relevant product documentation I wrote) that would allow a reader to accomplish both tasks. When I wrote code, I strived to keep the lines of code to a minimum. I wanted a reader to be able to run a model on an image in one section, for example. I provided that code, and some additional code that would let a reader see the results from their vision model on an image. Visualising results from a model is separate from deployment, but I considered it relevant because seeing results in computer vision helps a reader build conviction that a solution is working.

Here is an example of a code snippet from my guide:

```
import cv2
import supervision as sv
from inference_sdk import InferenceConfiguration, InferenceHTTPClient

image = "containers.jpeg"
MODEL_ID = "logistics-sz9jr/2"

config = InferenceConfiguration(confidence_threshold=0.5,
    ↪ iou_threshold=0.5)
```

²⁹<https://blog.roboflow.com/deploy-computer-vision-models-offline/>

```
client = InferenceHTTPClient(  
    api_url="http://localhost:9001",  
    api_key="API_KEY",  
)  
client.configure(config)  
client.select_model(MODEL_ID)  
  
class_ids = {}  
  
predictions = client.infer(image)  
  
print(predictions)
```

This is the minimum amount of code someone needs to run a model. I then explain how a reader can use the code, and what values need substituted (see the “Substitutions” section later in this guide for more information). In a separate section, I then elaborated on how to plot code, with reference to another minimum viable code snippet. I used two snippets, each with their own description, to:

1. Avoid having a very long code snippet, and;
2. Ensure that the main task (running inference on an image) was clearly distinct from an auxiliary but useful task (visualizing predictions).

I was judicious in what code I wrote, and didn’t write. I could have written more code to show more features of deploying a model offline, but I decided to stick with only what a reader needed to accomplish a task and verify that the code they wrote was working.

In total, I wrote 40 lines of code that show show to deploy a model offline. I could have written 100 with features like counting predictions for each class, adding exceptions to account for a range of edge cases. But this would have made the tutorial more complicated.

A reader should be able to comprehend every line of code you write that you say will solve a problem. The more code, the more a reader has to understand, and the more work a reader needs to do to figure out how to integrate your code into their application.

In short, keep it simple.

Explanations

My final tip is to explain, in text, what a code snippet does. Never assume the reader will understand what your code does without explanation. You may understand the code,

but you wrote it. In technical writing, your goal is never to copy-paste code with a few words that explains what problem it solves. Your goal is to help someone learn new knowledge and/or solve a problem. Achieving this goal requires detailed and articulate explanation on the part of the writer.

Your explanation might be a sentence that summarises the code or may be longer, depending on the audience for whom you are writing and what knowledge you can assume.

Readers can always skim over your writing if they only want code, but a significant portion of your readers will benefit from text descriptions of the code you include in a guide..

Here is an example explanation of a code snippet that I wrote in my offline vision model deployment guide:

When you run the script for the first time, the weights for the model you are using will be downloaded for use on your machine. These weights are cached for future use. Then, the image will be sent to your Docker container. Your selected model will run on the image. A JSON response will be returned with predictions from your model.

You should take extra care to explain details that may cause confusion. For example, you may note how long a code snippet could take to run if the code will take a while to run. You may point out explicit differences between two code snippets in your article if they are similar and have subtle but crucial differences.

Substitutions

Many code snippets cannot be copied verbatim from documentation. For example, you may need to change file names, URL names, or other information in a snippet to use it. You may need to add your own API key. Make sure you explain, in text, exactly what values need substituted and, where appropriate, where to find those values.

Here is an example of how I explained the substitutions required in the code snippet I referenced earlier:

Above, replace:

1. The image URL with the name of the image on which you want to run inference.
2. `ROBOFLOW_API_KEY` with your Roboflow API key. Learn how to retrieve your Roboflow API key^a.
3. `MODEL_ID` with your Roboflow model ID. Learn how to retrieve your model ID^b.

^a<https://docs.roboflow.com/api-reference/authentication?ref=blog.roboflow.com#retrieve-an-api-key>

^b<https://docs.roboflow.com/api-reference/workspace-and-project-ids?ref=blog.roboflow.com>

I stated exactly what needs to be substituted and where to find the relevant values. You should do this in your technical writing, too.

Sharing feedback

Through writing code snippets, I sometimes find myself thinking “there should be an easier way.” This is an interesting side effect of technical writing. When you write, you should strive toward making the reader’s experience as pleasant as possible.

If you are running into friction with code you are writing for a guide – for example, code that uses an SDK or a particular package – that may be a sign that there is feedback you could relay to an engineering team. If you think some code is so general that it could be abstracted into an SDK that your organization maintains, you could make that recommendation. If you find a bug in code, you can share that with the team.

That’s it for today’s technical writing tip! Join me tomorrow for another tip!

- Minimum viable code snippets
- Explain exactly what is going on
- Link to useful reading (how to find values for parameters)
- Can you abstract code upstream in a helper function / the function itself

Effective Examples

Earlier in this series, I spoke about the importance of referencing consistent examples in your work. I mentioned that it is acceptable to mention a few different use cases for a product or piece of software, provided those use cases are isolated in a particular section of your work (i.e. the introduction). You should focus in on one example when showing how to use the product, software, or code.

Examples serve as a means by which you can illustrate not only what your software does, but how it works. This can be wrapped in a narrative that helps build a consistent flow across your writing, easing readability.

What makes a good example in technical writing?

How do you choose what example to use in technical writing? That is a good question.

Good examples are:

1. Relevant to the capabilities of your software;
2. Well defined;
3. Intuitive, requiring little background knowledge about any topic in which the reader is unlikely to be familiar, and;
4. Relevant to your target customer (in the case of marketing-driven technical content).

Blog posts and tutorials should have a single, clear example that is used throughout.

Documentation that serves as the canonical reference for how to use a user interface should be example-agnostic, where possible. For example, consider the document titled “Create a Project” in the Roboflow documentation. Here, no example is needed beyond showing the product action. Text like “For example, you could create a project called ‘Cats’ that is used to identify cat breeds” would be superfluous and introduce more words than are necessary for the reader to accomplish the goal.

When I am working with team members who are writing blog posts, I am always available to talk through examples to help them find ones that meet the criteria above. My manager does the same for me.

How I used examples in a blog post

Let’s walk through an example blog post that practices the above advice.

Consider a guide, *Distill Large Vision Models into Smaller, Efficient Models with Autodistill*³⁰. This guide was an announcement for an open source project that featured a detailed guide on how to use the software. This blog post is consistently used as a canonical blog post on using Autodistill, the topic of the article.

Autodistill lets you use large computer vision systems that know a lot about the world to create smaller systems that can accurately identify particular objects. For example, you can use a large model that can identify 20,000 things to train a smaller model that identifies one thing (i.e. a milk bottle). The smaller model will run much faster and can be tuned to be more accurate than the larger model. Autodistill auto-labels images which are then used to create the smaller model.

In this guide, we opted to use the example of milk bottle and bottle cap identification. We showed how to create a computer vision system that identifies milk bottles and bottle caps on a manufacturing line. Such a system could be used in a bottling facility to count bottles at a given point on a manufacturing line, identify bottles that do not have a cap, and more.

Let's view this example through the four points that I said made a good example earlier:

1. Relevant to software capabilities: The example effectively shows the capabilities of the software.
2. Well defined: The example is clearly stated in the article.
3. Intuitive: Although the example is about milk bottles, you don't need to know anything about bottling facilities to follow the guide. Rather, bottles is used to illustrate the capabilities of the product, which is to auto-label images to make smaller, accurate computer vision models.
4. Relevant to the customer: This example is relevant to our expected audience, which is someone, ideally an engineer at an industrial company, who wants to reduce the amount of labeling time required to solve a problem.

When I write a new tutorial, I always think "what example would illustrate this product well, be interesting to our target audience, and is something that I can make in a reasonable period of time." That last point is important. Sometimes, I need to compromise when coming up with examples. If an example will take too long to create given the time available, consider if there are other compelling examples that would take less time to create.

³⁰<https://blog.roboflow.com/autodistill/>

Document limitations

When you are documenting software, make sure that you document limitations. Autodistill cannot auto-label any arbitrary image, for example. With that said, a reader could assume that this is the case, since we are introducing a concept that was previously used primarily in academia, and with which a reader in our intended audience may have limited to no knowledge.

Thus, we added balance, noting limitations in a “Best Practices” section. Here is what we wrote:

With that said, there are limitations to the base models supported at the time of writing. First, base models may not be able to identify every class that you want to identify. For more obscure or nuanced objects, base models may not yet be able to identify the objects you need to annotate (or may take extensive experimentation to find the ideal prompts).

If you are documenting APIs, limitations that you may note include:

1. Rate limits.
2. Properties that are not available in the endpoint documented but can be retrieved from another endpoint.
3. Whether an API is available in different SDKs (i.e. if an API is only available via HTTP, but not in your Python or iOS SDKs).
4. An API is about to be deprecated.

Where relevant, link to relevant resources about limitations.

Example bounds

You will also need to decide on bounds for examples. For example, I am documenting a Python library called `mf2py`³¹ with library contributors. This library lets you parse a HTML document to retrieve microformats data in a specific structure.

We decided to create a hypothetical HTML document with an author called “James” that was about Taylor Swift for a test to include in the README. This example was derived from a blog post I wrote, but was trimmed down significantly.

We didn’t think it was appropriate to use the exact post on my blog and reference it by URL, since the page could be moved one day and the URL breaking would make for a subpar experience. The page being down would be confusing if the post was broken and the reader used the post URL in the parse-by-URL example later in the

³¹<https://github.com/microformats/mf2py>

documentation. The output would not be as expected. In addition, the post I had on my blog was too long, which would result in an accurate but overly verbose output. So we trimmed it down.

After this discussion, we started to talk about the role of examples, and agreed to discuss guidelines that err toward more hypothetical examples, even if they are derived from content from the authors. (NB: This discussion is ongoing.)

Callout Boxes

Callout boxes can be styled in any way, but typically use the following hierarchy and colour scheme:

- Critical: A red box. Used for critical information. Use sparingly.
- Warning: An orange box. Used to highlight important information.
- Note: A piece of information you want to call out, but is not a warning.
- Tip: A handy piece of information that you want to highlight.

Let me talk through a few examples of these. *Note: I don't have an example of critical boxes to hand, but I do recall seeing a few so I wanted to add it above.*

Warning Callouts

You may use a warning callout to inform a reader that:

1. The API documented on a page is unstable. If possible, you should give a timeline on when the API will be final, and where the reader can go to check for updates.
2. An API has been officially deprecated. Such a callout should link to the relevant documentation for the new API.
3. A page is incomplete.
4. A page is under active maintenance and may change.

For example, I used a callout in a guide in Inference³², an open source computer vision inference server, to say that the model documented by the guide was not available. This box was added because we wanted to document that the model was being worked on so that users knew to check back for more information. Here is the callout:

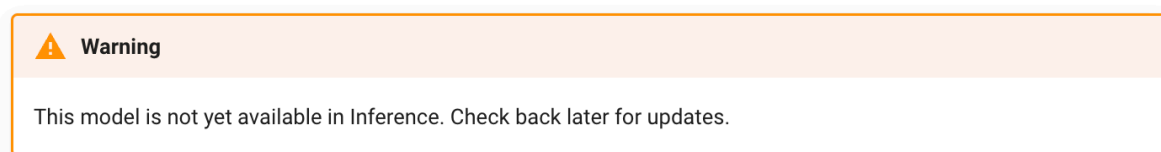


Figure 1: A callout box

Note Callouts

Note callouts are effective at highlighting information that you want to be highlighted but is not a tip, a warning, or a critical message.

³²https://inference.roboflow.com/foundation/grounding_dino/

You may use notes to:

- Highlight information about how different versions impact an API response.
- Call out that a task may take a while to complete.
- Inform the reader that they need to have an API key to follow a tutorial, with guidance on how to find one.

For example, I used a note to share guidance on the tasks at which a particular vision model performs well in a piece of documentation. While the information is close to a Tip, which I will document below, my guidance was not about direct usage. Rather, the information about what the model does well and does not do well could be used by a reader to evaluate if the solution in the guide is going to be useful to them.

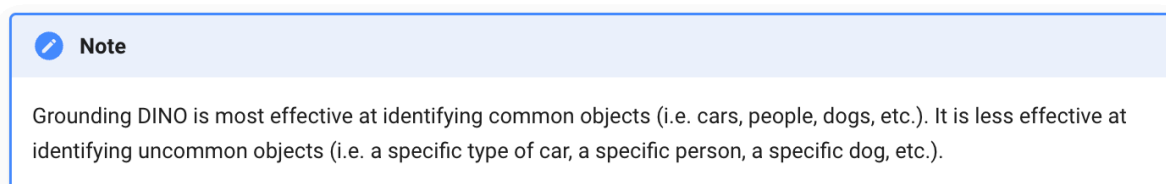


Figure 2: A Note callout box

Tip Callouts

Tip callouts are used to highlight additional information that helps a reader effectively use a piece of software.

You may use a tip to inform a reader that:

1. Reviewing a prior guide will give them the context they need to better understand the content of the page they are viewing.
2. A particular optimisation will help them save time.
3. A keyboard shortcut is available to perform a task.

For example, I have used tips to inform readers that a previous guide gives valuable context that will help them make the most of the page that they are viewing. Here is an example:

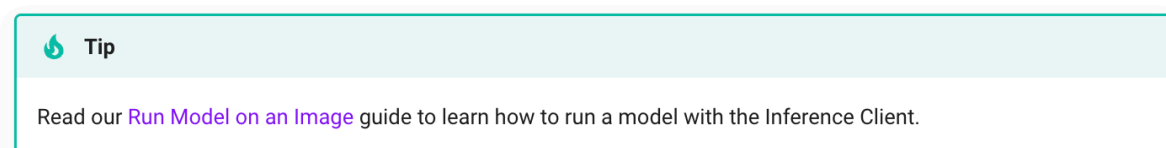


Figure 3: A Note callout box

That's it for today's technical writing tip! See you tomorrow!

Consistent Examples

It may be tempting to refer to many different usage examples for in a blog post. However, you should try to stick to one consistent example throughout a post and, optionally, have a separate section that lists additional use cases.

For example, consider a guide on how to search for frames in a video³³. This technology could be used on for media indexing, content understanding, and more. Throughout the post you should refer to one, consistent example.

I used to mention several uses in an introduction to a blog post, then potentially mention more in a section that defines required knowledge, before finally walking through one example in depth. It is easy to do this when a product is flexible, but many examples in different sections in an article – even if they are only one sentence long each – can be confusing.

Instead, I recommend:

1. Having a specific place where you mention examples. This could be your introduction or a dedicated “use cases” section.
2. Mentioning exactly what you are going to build in your introduction.
3. Following through with building that example application throughout the post.

In a guide I wrote on searching video frames³⁴, I showed how to search a movie trailer. Here was my introduction:

Imagine a search engine that lets you find frames in a video given a text prompt. For example, a search engine that helps you find all of the title credits in a trailer, all of the adverts in a broadcast, or all of the scenes that are in an office. With Roboflow’s video Inference API, it is possible to build such a search engine.

In this guide, you will learn how to use the Roboflow video Inference API to calculate CLIP vectors for frames in a video. We will then use these vectors to build a search engine that accepts text queries and returns timestamps for relevant frames.

I mention a few applications in the introductory paragraph. Then, I say that we are going to “build a search engine.” I then refer only to this example throughout the main content of the post.

Above, I spoke about referencing many examples throughout a post. I also recommend using one well-researched, robust example, and keeping it consistent. If you need

³³<https://blog.roboflow.com/search-video-frames/>

³⁴<https://blog.roboflow.com/search-video-frames/>

to show multiple product features, you should stick to the same example rather than introducing another example.

(Notice how in this guide I spoke only about the “build a search engine” guide, which I chose because I knew it would help me illustrate my point effectively.)

Run-on Sentences

Consider the following sentence:

On December 6th, 2023, Google announced Gemini^a, a new Large Multimodal Model (LMM)^b that is able to interact with and answer questions about data presented in text, images, video, and audio.

^a<https://deepmind.google/technologies/gemini/?ref=blog.roboflow.com#introduction>

^b<https://blog.roboflow.com/multimodal-models/>

Read it over again. Does it feel... a bit long? This is a run-on sentence, in which I have tried to string together too many facts in a single sentence. There are several commas in the sentence, some used to separate clauses and others used as part of a list or the date.

Now consider the following text:

On December 6th, 2023, Google announced Gemini^a, a new Large Multimodal Model (LMM)^b. Gemini is able to interact with and answer questions about data presented in text, images, video, and audio.

^a<https://deepmind.google/technologies/gemini/?ref=blog.roboflow.com#introduction>

^b<https://blog.roboflow.com/multimodal-models/>

Read the text above again. Does it feel a bit easier to read, and say?

In the text above, I made one small change from the previous one: I separated out the text into two sentences. Notably, the second sentence starts with “Gemini”. I did this to ensure the next sentence is clearly in reference to Gemini and not the LMM concept introduced in the previous sentence. While I can with greater ease infer that the second sentence is about Gemini, a reader not familiar with Gemini or LMMs may not be able to make that differentiation.

It can be easy to get in the flow of thinking about what you want to say and how you want to say it and lose sight of how long both sentences and paragraphs are taking. Longer sentences are hard to read, even if all of the clauses are directly related, such as was the case above. A balance between shorter and longer sentences is more comprehensible.

When you see long sentences, consider if you can shorten them. When you do, make sure there is an explicit connection between the two sentences. Look out for “This” and “It” at the beginning of sentences you write while fixing run on sentences. A shorter sentence is great, but if a sentence starts with “it” then you may accidentally introduce ambiguity. The reader should never have to ask themselves “but what does ‘it’ refer to?”

Duplicate Information

There is a general rule to which I attempt to adhere: Document once, reference anywhere. When information changes, you can change one place and know your documentation will maintain cohesion and stay in sync with the status quo of the product, API, or software you are documenting.

Pillar content

As part of writing and maintaining documentation, you will start to get a sense for “pillar”, or canonical, pieces of content. This content is your definitive source on a particular piece of information.

For example, Roboflow, my employer, has a blog post on image augmentations and preprocessing best practices³⁵. This is our canonical guide on the topic. When we write content that mentions augmentations or preprocessing, we often say that we recommend reading our full guide and link inline. We don’t repeat the same advice. We link to the relevant content.

By having key information in one place, you only need to update that information once if it requires a change.

Consider a guide that documents how to perform a product action, such as finding an API key. If you explain how to find an API key in every blog post you write that needs an API key, you have to go back and make those changes. If you instead say “Learn how to retrieve an API key.” in your content and link to your canonical source, you only need to update information once.

Keeping duplicate information in check

Duplicate information can be appropriate when you want to produce an end-to-end guide. For example, if you are writing a guide that shows how to get started with an application, you may repeat information that is available elsewhere. Having the information in a single guide is to the benefit of the reader: they can stay on the same page while accomplishing the goal of the tutorial.

With that said, you need to keep duplicate information in check. When you are writing, you should actively ask “is this documented elsewhere?” If the answer is yes, ask: should I be referring to that content? In cases where you find yourself likely to repeat paragraphs of information that is elsewhere, the answer is probably “yes.”

³⁵<https://blog.roboflow.com/why-preprocess-augment/>

Context and repeating information

There is a fine balance between providing context and repeating information. You may include a summary of a topic you introduce elsewhere in a guide as this is often beneficial to the reader.

If I started to write four paragraphs on what a topic is when I already have another guide elsewhere, the reader may be better served with a transition sentence to the effect of:

Interested in learning more about {topic}? Check out our full guide.

Here, I can point the reader to a canonical resource and ensure that the article I am writing contains only the necessary information the reader needs to achieve the stated goal of the article.

Duplicate information across platforms

In addition to duplicating information across content, you might find yourself duplicating information across platforms. This is presently an issue with which I am dealing. I co-maintain an ecosystem of libraries called Autodistill³⁶, a tool for automatically labeling images for use in building computer vision systems. Each library is its own Python package and GitHub repository. Every package is documented in two places: the package README and the central documentation.

The result? One source – the central documentation – is often out of sync, or missing pages entirely. We made the decision to have the information in two places so that no matter where developers came from – a README for a library in the ecosystem, the main documentation – they could find what they need. In hindsight, the READMEs were enough; we could always link to them in the sidebar of our documentation. That is what we are doing now.

Similarly, you might have information that is duplicated across a help centre and your main product documentation. When the underlying information changes, you need to update both places.

Often, it is not the act of updating the information that is arduous. Rather, it is remembering to update the information in the first place. To avoid this dilemma? Document once, reference anywhere.

³⁶<https://docs.autodistill.com>

How-To Outline

I often start my tutorials with a few bullet points of what I want to cover, or a few headings. These notes help me evaluate what I want to say in a piece of writing. By having a few notes, I am not starting with a blank page. The notes are the beginning!

With that said, you may be wondering: how do I know how to structure my article? How do I turn my notes into an article? For this post, I am going to focus on the “how-to”. I have prepared a few bullet points which outline a structure. You can copy-paste these bullet points into a document to get started, read them now, or save them for later as a reference.

The how-to outline

Here is the structure I would use for a how-to guide:

- How to do {x}
 - Introduction
 - * A few sentences of background.
 - * Introduce what you are going to do.
 - * Say what someone will be able to do by the end of the article.
 - * Link to related learning resources (a video, an interactive notebook (for data science, machine learning), full code documentation, etc.).
 - What is X?
 - * Introduce what X is.
 - * Provide context on X (history, when it was released, notable features).
 - Step #1:
 - * First, we need to...
 - * Include best practices.
 - * Here is the result:
 - [result]
 - Step #2:
 - * ...
 - Step #3:
 - * ...
 - ...
 - Conclusion
 - * For product documentation, a conclusion should be short. Perhaps a single sentence.

- * For blog posts and technical tutorials, a conclusion should be longer (1-2 paragraphs), summarising the content.

Walking through the outline

Let me talk through these bullet points with reference to an example.

Suppose I have been asked to write an article about how to deploy a computer vision model offline. I might start the guide with the problem statement: deploying a computer vision model is difficult and time consuming. Then I would introduce the solution, which would be our company's product. I would then state exactly what we are going to do "By the end of this guide, we will have done X, Y, and Z." If the guide has a visual end state (i.e. a completed action in the product or code), I may include a screenshot to show the reader early on what the guide will enable them to do.

The introduction should provide all the information a reader needs to evaluate if an article is likely to be relevant to what they want to learn or the problem they want to solve.

After the introduction, I would define the task we are trying to solve, with reference to the solution, in more depth. I might start a section that says "What is Roboflow Inference?", where I introduce our solution to the problem of deploying computer vision models and state the advantages of the solution. I could include multiple sections that talk about the task, depending on what information you think is relevant.

From there, we get into the "how-to" section. Create new sections for each step. I would start each step by saying what the reader will do, then show how to do it. Reference code where appropriate. Add images when you need them. Any images you add at this stage should be directly related to the "how-to" (i.e. product screenshots, code results³⁷). Create as many sections as you need.

Then, we can conclude. In a conclusion, you should summarize what you said earlier. Do not introduce new information in a conclusion. If you feel like you are repeating yourself, that is okay. A conclusion is an opportunity for you to help the reader reinforce their learning. Include key facts about the product / API you are documenting. Summarise the steps the reader followed to achieve the desired outcome.

If you are writing product documentation, I recommend keeping each section short and to the point. A few sentences – or even one sentence in some cases – may suffice for everything from the introduction to each step to the conclusion. Product documentation should strive to be as short as possible. Blog posts and technical tutorials benefit from a more detailed introduction.

³⁷If code results are in text, put the results in the article if they aren't too long. Screenshots of code results are not ideal; images take a while to load.

There is an expectation that a technical tutorial (i.e. how to use an API) contains learned insight beyond the exact steps someone needs to follow to solve a problem. For example, a blog post on “how to deploy a computer vision model offline” could explain the drawbacks of the current situation, with reference to experience. When you get a model working, you could show how to see the results (which involves more code.).

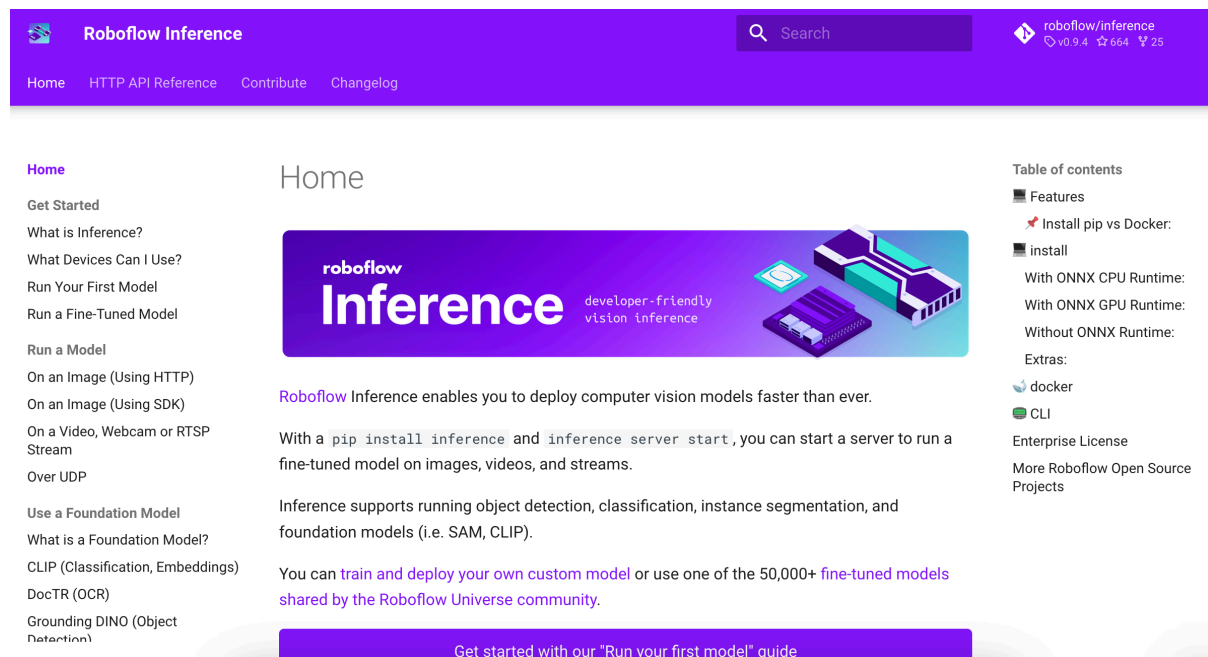


Figure 4: The documentation homepage for Roboflow Inference

Navigation Structure

I have been pondering about what to write today and I started to think about technical writing. As is the origin of many ideas, I combined my Advent of Bloggers series with technical writing (in general), to come up with a new series. Herein begins Advent of Technical Writing.

Over the next 24 days, I am going to share one lesson I have learned when writing technical documents. My experience pertains almost exclusively to software technical writing. I expect that much of what I write will be more applicable in that domain, but I hope my advice can generalise. This series will use a lot of “I”s, because I want to write with direct reference to my experience. You may have your own style that deviates from mine; therein lies the beauty of writing. But perhaps you will learn something from what I write!

This series is for everyone interested in technical writing, whether you are a beginner (like we all are at one point!), someone who writes a bit of documentation at work and wants to improve, an expert, or anything in between.

Without further preamble, let’s start the first edition!

Navigation Structure

Over the last year, I have led the development or restructure of several documentation sites, both at work and for personal projects.

As I write documentation, I pay a lot of attention to the navigation structure. This comprises a number of elements, including:

1. Whether you use a sidebar or a top navigation bar (or both) to allow people to find for what they are looking;
2. Deciding on the order in which documents appear in navigation;
3. Deciding on subheadings to organise content, and more.

I want to focus in on the second point above: ordering documents.

I commonly use a sidebar for technical documentation, in large part because the tools I use most for documentation sites (mkdocs and GitBook) both have well-designed sidebar features. When I write, or update, documentation, I always ask myself the question: in what order do I want someone to learn about the thing I am documenting?

I want someone to learn the “what is” as soon as possible, then provide a quickstart to get someone using the software. The quickstart should be use as little nomenclature as possible. When jargon is used that I cannot comfortable assume the audience will know, I aim to define said jargon. Quickstarts are effective because they are a single point to which someone can go to use your software; a reader doesn’t need to jump around to try and find an entry point. In most of my writing, the outcome of a quickstart is running a project on one’s own machine, but the aim will vary depending on the project you are documenting.

After the user has had a chance to experiment with the software through the quickstart, I want to graduate the reader into exploring more key value propositions that were not explained in the quickstart. Then, I go on to more advanced information, including reference material.

Let me talk through an example.

Consider Inference³⁸, an open source inference server that lets you run computer vision models on your own hardware. I started off by thinking about everything that I wanted to include in the documentation. Then, I divided the information into a few key sections applicable for the project. Here are the general sections of content I wrote, in order of how they appear in the sidebar:

1. A home page: The first place a person lands. I wanted this to be visual and provide a great summary to encourage someone to get started.

³⁸<https://inference.roboflow.com>

2. A list of “Getting started” guides which covers one of the main value propositions, and the topic that I decided was the most appropriate entry point: running a fine-tuned vision model on one’s device.
3. A “Run a Model” section with guides on how to run a model. Each tutorial is titled “On a...”, such as “On an Image (Using HTTP).” With the sidebar heading as context, each link concisely states exactly what each guide does.
4. “Use a Foundation Model”, which documents the foundation model features in Inference. These features are important, but are more effective to read once you know about how to run a model.
5. “Integrate with Inference”, which includes more advanced documentation.
6. “Reference”, which includes miscellaneous reference information that someone may use with the library.

This sidebar appears on the homepage, and all of the pages linked in the sections mentioned above. The sidebar changes if a user navigates to a different area of the site using the navigation bar at the top of the page. The link “Home” always appears in the top navigation bar, allowing a user to get back to the above documentation, which covers the most important parts of using Inference.

My goal for the documentation was to achieve the following in order:

1. Tell someone what the product is.
2. Tell someone how to use it as soon as I can.
3. Highlight key value propositions, in order of relevance to a beginner:
 1. Running a model on images, videos, webcams, and streams.
 2. Using foundation models.
 3. Using the SDK integration.
4. Including important reference material that users will need.

The structure above assumes you are documenting a single product. For a site that documents multiple products, you may choose to have different subheaders in a sidebar for different products, or different links in a navigation bar at the top of the page that takes people to different product home pages or guides.

I like Supabase³⁹’s structure, where they have a parent sidebar and child sidebars for each product or topic area. For example, “Database”, their flagship product, is at the top of their “Product” sidebar section. This section is directly below their quickstart. If I click into “Database”, the sidebar is replaced with guides related to their database product. I can always navigate back by clicking the “Back to Home” link that appears.

Generally, be conscious that document order matters. Order guides in such a way where a new reader can easily find for what they are looking.

³⁹<https://supabase.com/docs>

Join me tomorrow for another tip!

Navigation Links

You may be wondering: “what constitutes an ‘effective’ navigation link”? An effective navigation heading is one that:

1. Is short, ideally no more than four or five words. How many words you can use will depend on if the link is in a sidebar, a dropdown, or a top-level navigation link. In general, use as few words as you can.
2. Summarises exactly what a user will learn to do. For example, “Managing indexes” or “Create a Project” or “Search a Dataset”.

Consider the following three examples:

- “How to export data”
- “Export data”

While both are descriptive, the second link is more direct. “How to”, when used across many links, will be repetitive. Navigation links in documentation already imply that a user is going to learn how to do something, or acquire the knowledge they need to make a decision or solve a problem. Thus, words like “How to” are often superfluous.

If a user could export different types of data, and the flows are different, we could have a subheading in navigation called “Export”, with links for each type of export. A link may only take the anchor text of the name of the data to export, since the heading would give information. Here is an example:

- Heading: Export Data
- Links:
 - Images
 - Annotation Statistics

Here, we don’t need to say “Export Data” every time, since it is implied by the heading.

Links should be under a relevant heading. Here is an example from some documentation I wrote on deploying a computer vision model, under the heading “Datasets”, a product feature:

- Create a Project
- Upload Data (expands to specific ways you can do this)
- Search a Dataset
- Create a Dataset Version
- ...

Every heading is direct and action-oriented. To the extent that you can reduce ambiguity in documentation, do. Effective navigation links go a long way in helping the user find exactly for what they are looking.

This advice pairs with the first guide in the series, in which I talk about how to structure navigation links.

Technical content and a-ha moments

Earlier today, I was working on a piece of content and I thought to myself: if the only part of this guide that someone reads is the first few sentences, are they going to be enticed to keep following? Would they have the information they need to understand whether the rest of this article is for them? To set the bar even higher, consider this: what if the reader is skimming the content, as one so often does when evaluating whether a piece of content will help them solve their needs.

I like to explain directly in the introduction of a guide what that guide will discuss, either in bullet point form or a few succinct sentences. I add line breaks liberally to make the content easy to skim. If available, I'll add a video or an image at the bottom of the introduction with the idea in mind that I want to show, not tell what I am going to guide someone through doing.

This applies not only to text content, but video, too.

Earlier I was preparing to record a video and I was thinking about the sequencing of that video. My mind drifted toward a concept that was introduced to me in the opening lecture of the online FastAI machine learning course. The instructor wanted to show what you could do before explaining every little detail. I, too, want to do this in my videos. I thought to myself earlier today: I should put the demo at the front-and-centre.

Don't bury the lede. Concisely summarise the key points, then delve deeper. With this mindset, I find myself trying to refine the "a-ha moment" a bit more in my head. Often, that moment doesn't come before writing a piece. It comes during the writing phase or after an editorial review. Indeed, one of the best pieces of feedback I have received is, to the affect of "we need to make the a-ha moment clearer." As a reader, too, I appreciate it when the real utility of a product or software, the exciting part of a product launch, or the goals of a technical guide are concisely and prominently defined.

Feature Releases

In writing, you will become an expert on the feature and may come up with ideas that could be used to improve the product. You will push your team to have answers to key questions, like for whom is a feature available, which will help the team become aligned on a new release. Customers will benefit from comprehensive, cohesive knowledge that acts as both an announcement and an introduction to a new product.

When I am writing a feature release, I aim to answer the following questions:

- What is the feature?
- Who would use the feature?
- For whom is the feature available?
- How can the reader use the feature?

What is the feature?

Defining a feature is an essential part of a feature release. Let's look at how I announced Video Inference, a new product release from Roboflow. The blog post covers the product itself (the video inference infrastructure) and the means by which someone uses it (the SDK).

I introduced the product like so:

We are excited to introduce the Roboflow Video Inference API^a, which allows you to apply computer vision models to videos in the cloud. With the video inference API, you can use the video infrastructure we've relied upon internally, which handles things like batching requests, concurrent processing, and complexity associated with scaling with inference on the frames in a video.

^a<https://docs.roboflow.com/deploy/video-inference?ref=blog.roboflow.com>

How you introduce a product is very much a marketing question, but refines a muscle you will use all across your technical writing: explaining concepts with clarity, adjusted for the audience.

Above, I say what the feature is and what it allows the user to accomplish. This information is included in the introduction, after some contextual information that helps guide the reader in the guide.

Who should use the feature?

Product and open source releases are rarely relevant for everyone in your audience. You should clearly state who would benefit from using a feature. You can do this both

by stating key features and by providing use cases.

By stating features, readers can evaluate the product or code. For example, I noted:

The Video Inference API is designed to scale with any use case, whether you are working with a small collection of videos, an archive of years of material, or videos that come in every day. Use secure signed-URL endpoints to upload videos from S3, GCS, or Azure Blob Storage with confidence. All this makes adding video inference a secure and frictionless experience.

To the audience I had in mind – people who want to use computer vision models on videos – this is important information. Knowing that the solution scales and works with cloud storage providers is key context that speaks to how the tool might fit in your workflow.

Let's talk about use cases. Another way to think of a use case is an issue for which your product or code solves. In my video inference guide, I made a list of a few use cases:

1. Generate tags for use in building media search engines;
2. Analyze videos for brand and safety requirements;
3. Run security footage to detect objects of interest;
4. ...

This list of use cases explicitly outlines some of the many ways in which the tool could be used.

For whom is the feature available?

State for whom your software is available, especially if there are limits on who can use the software. Does someone have to opt in to a beta in your product? Is the software available only for paying customers? Is the software generally available?

You should state if any of these cases are true with the requisite context. For example, if a feature is in beta, you may want to set expectations of reliability. If a feature is only available for paying customers, you may want to include how someone can use the feature (i.e. reach out to their account manager).

How can the reader use the feature?

Once you have announced the feature, you should show how to use it. This is your time to walk through an end to end example of the software or code in action. In the

aforementioned video inference guide, I showed how to use the feature in a way that is intuitive. The reader can start with a video and end with predictions visualized on a video. The guide could have ended by showing the JSON results from the API, but showing how to plot predictions will help improve comprehension over what the API can help you achieve.

Conclusion

The advice above applies no matter if you are writing a product announcement, an SDK extension, a new feature available in open source software, and more. By answering the four questions above, you can ensure that your readers have the information they need to:

1. Know what a feature is
2. Evaluate if the feature may be useful to them
3. Experiment with the feature

Of note, a product announcement should not be the same as your product documentation. Whereas you can walk through an example use case in a product announcement, product documentation should be use case agnostic and document all of the edge cases and usage situations.

The aforementioned video inference guide walks through an example of sunning an object detection model on a video. But, the tutorial notes other models are supported, such as foundation models like CLIP. How to use other models is explained in the full product documentation, which is linked on the page. A product release should not be the canonical reference material for a feature, rather an introduction to that feature.

That's it for today's technical writing tip! Join me tomorrow for another tip.

Authoring Tools

Directing contributors to use specific tools

We ask contributors to use specific tools for two reasons:

1. It reduces ambiguity in the publishing process, and;
2. We, the content team, can ensure that all content is contributed in a tool in which we are able to provide collaborative feedback.

Reducing ambiguity in publishing is essential. To the extent we can keep the process standard, we do. At Roboflow, we used to allow people to write Google in our CMS directly or in a Google Document. With no clear directive, content was delivered in both tools. Because our CMS has inferior editing tools to a Google Document, providing feedback was harder. This was a significant issue. Our editing process is hands-on; we strive to give people detailed feedback on their work. Our CMS limits our ability to do this.

Indeed, CMS's are mainly designed for *publishing* not collaborative authoring. By opting for collaborative tools where possible, we, the content team, can be more expressive and detailed in our feedback.

Tool choices

For blog posts, Roboflow asks contributors to author documents in Google Documents ('Google Docs'). The editing interface provides an extensive set of features that overlap with what we need for our blog posts. Furthermore, Google Docs provides a strong set of collaborative features that are important to the content team. Multiple people can edit a document at once, reliably. We can leave comments inline that can be accepted or rejected with the click of a button.

For product documentation, contributors are asked to work directly in our documentation tool. This is because changes can involve a lot of custom formatting. For example, a contributor might add API specifications, code snippets with multiple tabs, etc. Rather than write in a separate tool, we empower team members to write directly in the documentation so they can mark it up as necessary. We ensure all team members have access to our tool if they want to make a change, provide guidance on how to make a change if required, and answer any questions along the way.

For open source documentation, we collaborate on our documentation in GitHub. We ask that documentation is provided with new feature changes. Documentation is a requirement for shipping. I usually do documentation reviews and provide comments

using the PR review tool available in the GitHub web interface. I leave all my suggestions as “Comments” to ensure that my editorial notes do not constitute approval for the code in a PR. This is essential. Documentation review is not a code review.

These are our tool choices, but the tools you use may vary. Indeed, you should choose the tool that works best for your team. Importantly, the tools in which content is delivered to an editor should empower editors, for editing is their role. If a team member writes a document in a tool with which an editor is not familiar, the editor will likely have to move the contents into another tool. This is less than ideal, introducing the potential for content to be accidentally modified during the move between tools.

Deprecating Content

Communicating with the team about deprecations

Deprecations rarely come without notice. As a technical writer, you should stay close to the teams whose work you are documenting. Listen out for any code, SDK, or product changes that may require action. When you hear that a tool may be deprecated, start communicating. Gather information on:

1. What is being deprecated;
2. When it is being deprecated;
3. Why it is being deprecated;
4. The extent to which the old software / API will be supported, and;
5. Any transition plan that other stakeholders (i.e. product or engineering) have in mind.

At work, we are presently deprecating a video stream interface for one of our open source products, Inference⁴⁰. This work had been going on in the background; it was on my mind. Then, as we got closer to release, an engineer wrote documentation on the new interface. I started to mentally collect answers to the above questions. Let me answer the questions above, in order:

1. An API, called `inference.Stream()` was being deprecated.
2. We were announcing it in our upcoming release in a few weeks.
3. It is being deprecated because we have made a new version that is significantly faster (as evaluated by benchmarks) and includes helper functions that reduce the amount of code our customers need to write.
4. The old API will remain supported. But, we will encourage all users to use the new one.
5. We have no specific date in mind for the feature to be officially dropped, but when that date comes we will give sufficient notice to users and update our documentation as necessary.

Having this information was immensely valuable. I could start to think where we had documented the feature, the urgency with which I would need to update content external to the main documentation (i.e. tutorials and blog posts), and keep in mind the new API as I wrote new content. I could also make sure I recommend it to our field engineering and sales teams if I get questions. I could think about promotion.

⁴⁰<https://inference.roboflow.com>

The deprecation lifecycle

Content deprecations start with an announcement that a product, SDK, or API is being replaced with a newer version, or will no longer be supported. You will likely need to produce content on both the new product or API being introduced and work with your team to put together a deprecation plan for content that is about to go out of date. The deprecation (i.e. setting timelines, removing code from a codebase) will be managed by product, but you will be responsible for the communication work in terms of updating content and writing new content where relevant.

When you start planning a deprecation, take notes on:

1. All of the places where content will need to be updated;
2. Site pages that you may want to deprecate;
3. What new content you will need (product pages, blog posts) to ensure a new feature is well documented, and;
4. The order of priority in which changes and new content needs to be made.

When it comes to new product or open source changes, my first priority is the product or code documentation. Since the main product or open source project documentation is the canonical source of documentation for a respective project, it is important that the requisite updates are made there, first. Then, I work my way through other places that need to be updated (i.e. blog posts).

Once you have made the requisite updates, you need to communicate them to your team. Make sure everyone knows about your new documentation so they know to refer to it. This is particularly important for significant changes that may effect the architecture of a site (i.e. major API releases that are available on a different subpath).

Below, I am going to talk in more detail about deprecating APIs and content (i.e. blog posts). I am not going to talk about deprecating full pages of product documentation since I have less experience with doing so.

Deprecating APIs

When you deprecate an API – whether it be a REST endpoint, an SDK function, a CLI command, or something else – it is important to have a plan in place. When you deprecate an API, you will have two pieces of content to think about: the documentation for the old API and the new one. This documentation may be on the same page or across different pages.

Your old documentation should include callout boxes that informs users of the deprecation. You should state the official date after which an API will no longer be supported with as much advance notice as possible.

Your new API should be prominently located in your documentation when a user is looking for the feature. For example, we are updating our quickstart guides to use the new Inference streaming interface. We will ensure the new one is high up in our `Run inference on a video` guide, which is the canonical source of information about the API. We will include a note about the old API being deprecated. We will keep our content on the old one for users who need it, but that content will be in a tab or toward the bottom of the page.

For more complex projects such as introducing entirely new APIs (i.e. a GraphQL API), you may opt to use documentation versioning. This involves having separate sections of your site (i.e. `v1`, `v2`) for each API.

Ensure planned deprecations are announced in advance in release notes and change logs, and when an API is officially deprecated or unsupported.

Deprecating content

In addition to deprecating APIs, you may need to deprecate content if the underlying product or API on which it depends has significantly changes. This could involve updating help center articles, blog posts, product guides, or other materials that you have written.

For example, Roboflow released a new Video Inference API, which enables you to run inference on videos in bulk faster and cheaper than using our image API, which was designed for images. The team wrote a guide two years ago on how to use video inference with our old API, which is now out of date. We wrote a new guide on how to use our new API since the methods are fundamentally different. We will redirect our old guide to the new one so that users can easily find it.

It is important to make sure changes are made across all sites you own when you deprecate content, not exclusively in the documentation most directly related to your product. For example, our old video inference blog post was not in our product documentation. But, it ranks in the top spot on Google for `roboflow video inference`. If people search for this product in Google – a likely user journey – they would see old information.

If you think a piece of content is no longer relevant, you may:

- Un-publish the old content and redirect to a new page with up-to-date content, or;
- Update the old page with a clear disclaimer notifying the user that the content is deprecated.

In cases where you have a new product or guide that a reader can follow, un-publishing the old content and redirecting to a new page is the best experience for the reader. This

approach eliminates the need for a reader to make a decision to navigate elsewhere themselves to find a solution to their problem. If you are deprecating a feature and dropping support, the latter option – adding a disclaimer to the page that the content is deprecated – is ideal. In that case, you should still refer to other helpful resources that may assist the reader in finding the knowledge for which they are looking.

Conclusion

Good documentation is accurate, up-to-date, and provides the reader all the information they need to learn about a subject or accomplish a goal. Documentation is not static, however. As a technical writer, you will create content, design navigation structures, redesign navigation structures, update existing content, and deprecate old content.

When you hear that an API or product is going to be deprecated, make a plan with the relevant engineers and product managers. Ensure readers of your documentation have sufficient notice, where “sufficient” is determined by your team.

For big changes, like an entirely new API version, you may launch the new API version and maintain support for the old one for a few years. For small changes to an SDK, you may officially drop support for a function or a method after less time. Whatever the case, make sure readers know what deprecations might affect them. Update documentation as appropriate. Redirect old documentation to your new documentation.

Facilitating Ideas

Ideating

A significant portion of the ideas on which I work come from discussion with the rest of our marketing team.

We regularly check in on updates in the computer vision industry in which we work. This is essential because the industry is changing week-by-week. Sometimes our priorities can be shifted for a week so that we have time to create content on new topics that would be relevant to our audience of people – students, customers, enterprises – who turn to us to learn about the latest in computer vision.

We also discuss what is going on within the organization. Indeed, a technical writer should be integrated into as many parts of an organization as possible. The more I know about what other teams are working on, the more effective I can be at supporting the broader organization's goals

For example, last week our team spent time on producing content on a new product, Video Inference. The more content we have on the product, the more resources we have that can be sent directly to customers who may benefit and promoted.

Although the marketing team comes up with a lot of ideas, in many cases ideas on what to write come from other departments. Sales. Customer support. Senior leadership. Product. Field engineering.

I recommend that technical writers make themselves as known to an organization as possible. Position yourself as a resource; be there to support, advise. Invite people to contribute blog posts. Offer to write blog posts about topics that other people are interested in. Everyone who works at your business may have ideas on what content would be useful to customers, users, and other stakeholders.

For example, we recently finished a long-running technical content marketing initiative with various industry partners. This idea initially came from senior leadership, then I discussed the idea with our marketing lead (head of department). We noted some content ideas and over time I contributed content to the initiative. I ended up writing several blog posts that integrated directly with a core product.

Last week, I wrote a guide on deploying vision models offline. The idea was recommended by the head of one of our product lines who noticed we didn't have content. We published a guide on quality assurance, using skateboards as an example. This was written by a new team member who used his years of experience in manufacturing to inform the post direction.

Ideas come from everywhere.

Prioritising

We have a Miro board at work on which we take notes for potential content ideas. Miro is effective because it is visual and interactive. You can arrange tasks however you want. Our marketing team doesn't assign deadlines for topics, rather expectations that certain content should be completed in a given week, or may be worth exploring at a future date.

To prioritise work, we ask a few questions:

1. What content will help stakeholders solve problems?
2. What content have team members suggested we produce?
3. Are there gaps in our current documentation that need filled?
4. Is there feedback that we need to act on?

From there, we come up with themes for a week. Last week's theme was Video Inference, a new product, as well as wrapping up some content marketing with industry partners. This week's theme is multimodality, a concept that refers to machine learning models that can work with images and text to solve problems, as well as general documentation cleanup.

I recommend working across themes in your team to avoid context shifting. Set goals around a specific product or an initiative that your team – and other stakeholders – collectively agree is a priority. I rely a lot on our marketing lead to help steward prioritisation discussions but I always provide insights to help us agree on priority. Indeed, prioritising should be a conversation.

Most of our work is prioritised a few days or weeks in advance, depending on the task. However, we always have room for content that pops up that requires immediate attention. For example, we respond to fixes suggested by other departments in a timely manner. We cover industry trends with new technical content. We lend an insight into how we should position a piece of software while there are still active product discussions going on.

Plates: Don't fill them too much

Every week, you should work toward defining a clear scope of work that you think is achievable. Don't fill your plate too much. What constitutes "achievable" will be based on all of your past experience, as well as the information you have about the initiative on which you are working. This is a muscle that you will exercise every week.

When I see priorities of the week, I take notes of what I think I can do and suggest some content that I think will be difficult to fit in the week. I also think about balance.

If I have had a week of writing a lot of content, I may ask to focus on other tasks, like contributing features to our open source software based that are open and do not have an owner.

Writing documentation is a long-term game. Split large projects into smaller, achievable tasks. While the prospects of a new project may be exciting, it is important to set realistic expectations for what you can work on and by when. Anticipate that other tasks will come up and use that to inform to what you can commit. Someone may ask for your support editing a blog post, you may be asked to sit in on a few product calls, you may see an opportunity for learning about something new that you want to seize. You want to be able to be present.

For example, a documentation overhaul project in which I participated earlier this year took around a month to be considered complete. We allocated a week for internal feedback, a week for talking with customers and synthesizing feedback into action items, then two weeks (with room for more) to act on feedback. This was a marathon, not a sprint. In the end, the project came out really well, but I needed all the time I had to deliver on the task.

Being explicit about what you can do each week is helpful to everyone. Your manager will have more information to evaluate how much work they can assign. You can better plan what content everyone on the team will write. You can more confidently say that certain projects that have external stakeholders will be delivered on time. Expect your motivation to flux. We can't write words all day, every day. But that's okay! For, as I have documented, the role of a technical writer is about so much more than words.

Internal Code Documentation Requirements

What does it mean that “documentation must be written on new features”? For supervision, this means that:

1. You must have docstrings in your code that clearly define new functions and classes.
2. Docstrings for functions *must* contain a working minimum example of how the code could be used.
3. You add a reference in our `mkdocs` site to auto-generate documentation from the docstrings where appropriate.

If you are adding a bigger feature, you may need to add another page of documentation. Because the library is more mature, this is not the average case. New APIs are introduced incrementally. When they are, documentation is written.

Another project to which I contribute, Inference⁴¹, is also working toward making documentation as a “requirement for shipping.” That is to say that you should have documentation for the feature you have written. At minimum, you need to write documentation that says how to use what you have written. The documentation doesn’t need to be long, but it does need to be comprehensive. Alternatively, an engineer can write a few notes or record a Loom video (a screencast) and I can turn it into full documentation.

You may be wondering: how do I get engineers on my team to sign up for documentation as a requirement for merging PRs? Will it slow my team down? The answer to that? It will take some time to write documentation, but in the long run, you will save time. This is also the case with PR reviews. It takes some time to review PRs but by having a robust PR review practice in place you can mitigate bugs and help engineers learn about a codebase and grow their skills.

I suggest keeping documentation requirements minimal; (accurate) docs are better than no docs, even if the docs may need work later. Don’t expect perfection. You can always refine documentation over time.

To get buy in from your organization, here are some points you could share about why writing documentation before pushing (or around the launch of) new features is useful:

1. Your marketing team has more information that they can use to promote your software (this is particularly important if you are writing open source software and growth of said software is part of your business).
2. Your sales team knows exactly where to point customers and prospects who have questions about a new offering.

⁴¹<https://github.com/roboflow/inference>

3. Customers will have a more pleasant experience using and integrating with your feature.
4. Internal team members working *with* the project should not have to ask as many questions getting started, saving time for the engineers who would likely answer those questions.
5. New team members working *on* the project will have more context about its main parts.

For example, you could start by saying that all new Python functions require docstrings in PRs and those docstrings be referenced in your main documentation. You could say that large features that are important to your customers should have their own tutorial, but this could be co-written with a technical writer if required.

Even if documentation is not a requirement for shipping features, your actively encouraging people to write documentation will go a long way to helping create a culture where more people document code. It might be the case that your mentioning documentation a few times causes it to come up in sprint planning, with tickets or notes left on the board about important product features that need documentation.

Internal Dry Run

I joined the call he was recording, staying quiet and only providing assistance when directly called. Because the team member had never used the project directly before, this was what I would call an “internal dry run” of the documentation we had. In the process, I took notes on the points of friction, which I could summarise and act on later.

At the crux of the “internal dry run” is:

1. Inviting a team member to try to accomplish a goal, with reference to documentation.
2. Asking them to take notes or, ideally, to record their experience.
3. Summarise feedback, then act on it.

I like this methodology because you get to see how someone who is not involved in a project uses it. Indeed, it is better if a team member runs into an issue with your documentation than it is a customer.

When you write a new piece of documentation, you may have it edited by another team member (I say “may” because editing resources are limited on smaller teams). This edit only one step in assuring the quality of your documentation, a topic I expect to talk about in a full post. But no matter how much quality assurance you do, there is no substitute to someone using your documentation from scratch to solve a problem.

In an internal dry run, your team member might find:

1. Missing dependency installations (your development environment will be different from that of your colleague).
2. Topics that aren’t explained well for someone not familiar with the project.
3. Content that should be more prominent in the site navigation structure.
4. Software bugs (which you would then report to the software maintainers internally, if you are not a direct code contributor).
5. And more.

This internal dry run caused me to restructure the documentation for a project and dedicate time to writing new guides and tutorials. In the end the team member who did the dry run reported that the edited documentation was an improvement over the last. A win indeed!

You may be wondering “when is the best time to do an internal dry run of your documentation?” For a single page, you can do these regularly, although perhaps without the recording component. For a dry run of an entire project, you may do them less regularly, perhaps before a project is live and any time you make significant changes based on feedback.

Of course, the dry run is no substitute for feedback from users, customers, or whoever the audience is for your content (this could be an internal team, for example, if you are documenting an internal tool). But, it is an effective way to find issues. I plan to do one for an upcoming improvement to some documentation!

Holistic Documentation Reviews

Every so often, I recommend taking a step back from the projects on which you are working to ask: is this documentation, as a whole, achieving its goals?

This is a question I asked myself earlier this year when working with the Roboflow product documentation. The documentation was complete and comprehensive, but we had built up some “documentation debt” per se. For example, some screenshots were out of date. Some sections were not ordered as intuitively as they could be for new users.

My analysis, and the growing general sentiment that we could improve our documentation, led to a holistic documentation review.

Committing to a review

In my day to day work, I started to see improvements that we could make to our documentation. These improvements would help us better position our products with our new products in mind, aid in users finding the information they need, among other things.

If you think documentation could be better, start by flagging it to the relevant party (product owner, technical marketing manager, etc.) that you see opportunities to improve a documentation site. Reference specific instances where you think the site could be improved. Have a conversation. It may be the case that your manager and potentially other people in the organization feel the same way about a review as you. If there is interest in doing a holistic review, share that you are interested in taking lead on the project.

What to ask, and how

We started the Roboflow documentation review with two main questions:

- What are we doing well that we should do more of?
- What could be improved?

I started by writing a document that listed suggestions, and then worked with the team to facilitate feedback. It is important that you not only ask the right questions, but that the right people are involved. In the case of this project, we invited the whole company to participate.

Indeed, documentation is something in which every team has an interest. Accurate, comprehensive documentation helps customer support have better resources to which

they can point users. The product team can confidently state that there is documentation on how to use what they have built. The sales team can confidently refer to documentation that shows how to use our product to solve a problem in customer calls.

We collected feedback for a few days, as well as ideas on how we could improve our documentation. I proactively invited people to participate, with a Google Doc in which suggestions were noted. I invited comments on existing feedback, too. The more input, the better. Getting people who work directly with customers involved on a regular basis was important to me. They could note common questions they had heard that could be used to inform improvements.

I set a specific date after which feedback would be collated and implemented. I was always open to more feedback, but setting a deadline helped me ensure that people knew by when I expected input.

Here were some of our findings:

1. The structure of navigation links was not linear, which made it hard to find some information;
2. Some documentation could be simplified;
3. Text was styled inconsistently across some pages (i.e. some pages used bullet points liberally);
4. We had new products that we wanted to ensure were visible, among other findings

At this stage, we knew both that we could improve our documentation and had an incline as to areas for improvement. Some people left excellent ideas that we could use to inform tactical changes, too.

But there was one thing we needed to do before making significant changes: user research.

User research for documentation reviews

Internal stakeholders have extensive knowledge that is useful in preparing documentation. For example, product owners know about the features available in a product and how to use them. With that said, there are many things that internal stakeholders cannot tell you because they work with your product or documentation regularly.

That is where user research comes in. For this project, I reached out to several customers and booked three calls in which I asked variations on the same two questions with which we started our review:

- What could be improved with our documentation?

- What do you like about our documentation?

I wanted to focus each call on issues rather than getting caught up in trying to solve a specific problem on the call.

In exchange for the time participants gave to answer questions, I offered to answer any questions about our product and provide support where needed.

Through this process, I was able to gauge how external stakeholders felt about our documentation. I was also able to discover specific ways in which our documentation could be improved to help them.

Scoping

With feedback from internal and external stakeholders documented, we were now ready to start defining a scope for changes. This scope would dictate how much time we wanted to spend working on different areas of the documentation.

We agreed that rewriting pages was in scope, where pages needed substantial improvements. We agreed on editing pages to bring them into a consistent style. We decided to revisit the navigation structure, especially the Deployment section of documentation (an area for improvement highlighted by the field engineering team).

When you scope changes based on a documentation review, you should aim to address as many of the areas for opportunity that you identify. With that said, the amount of time you have available will necessarily limit what you do.

One prominent discussion was not about the documentation itself but rather the tool we used to host our product documentation. Alternative tools offered features like more robust support for parsing OpenAPI documentation and auto-generating pages for endpoints as well as more flexible navigation structures. Ultimately, we decided not to change tool because we could get more from our existing tool with some changes, but to keep this in mind for the future.

Making the changes

Now came the fun part: updating the documentation. This was and is a heads-down process. I read over pages across our site, making edits where appropriate. I added stronger first sentences. I removed obsolete sections. I worked with teams on sections where I required more information to make edits.

Across a site with 70+ pages, I spent around a week of focused time working on changes. Most of my role was editing, with minor changes. In some cases, I wrote new pages.

One of the highlights of the experience was addressing a deficiency in how we documented our deployment solutions, which are used to run computer vision models on hardware like a Raspberry Pi or an NVIDIA Jetson. Up until the review, we were focused more on the *how*. We did not have as much contextual information that would be useful as a reference. We identified a few new pages to create, which included information on:

- A matrix of deployment options and what models were supported by each option, and;
- Models whose weights you could upload to our platform, among other topics.

We also used this time to revisit our API documentation. While our routes were documented, we re-ordered the documentation so that routes were roughly in order of the typical user's journey. For us, this meant we featured "workspaces" (a group of users) before "projects" (that are associated with a workspace), which were both before API endpoints on topics like model training and using a model. Adding linear progression to the navigation structure makes it easier to find the endpoint for which you are looking.

I am not going to list everything we did, but some other tasks we worked through included:

1. Adding new screenshots
2. Updating existing screenshots where our user interface had changed significantly
3. Updated the homepage to link to relevant resources
4. Deprecated documentation that was no longer relevant
5. Moved long scripts into their own GitHub repository
6. Added documentation for more Python methods
7. And more

We didn't incorporate all feedback. We prioritized feedback based on time required and expected impact for customers.

Announcing the project, and review

With the changes ready, we announced internally that an updated documentation site was coming soon that improves on our existing library of documentation. We shared the site for everyone to look at and launched it to the world. This was the exciting part: the culmination of weeks of thought, review, discussion, writing, and editing.

We have heard positive feedback from customers (which team members have kindly highlighted to me!) about how our documentation helps them use our platform. We received praise internally for the project, too. After working on this project, I feel more confident in our documentation being easier to navigate, up to date (at the time of

editing), and consistent. This was important to me because, as a writer, I could see opportunities for improvement that I wanted to address.

Since this project, I have led two reviews for open source projects. Both reviews were on a much smaller scale. One was prompted by a dry run (which I recommend everyone try with documentation!) and the other was prompted by internal friction around how to use an open source package. In both cases, seeing someone experience friction with a product ignited us to action. As a technical writer, it is your responsibility to be the person that takes friction and turns it into tangible results.

Through holistic documentation reviews, you can catch the little details that you might miss in your day-to-day. Through a holistic review, you will identify opportunities for improvement that make your documentation more usable, intuitive, and complete for users.

Publishing Contributor Blog Posts

In this post, I am going to talk about a few tips for managing internal contributors' work. I will focus on internal contributors, since managing content written by third parties is a separate process. I plan to document said process in another blog post.

Throughout the content production process, I explicitly communicate to internal contributors that I will manage the publishing process. I do this when we first start talking about the idea for content and after we have worked together to arrive at a final draft. This is important to me because I want to make sure contributors know that they do not need to learn a new tool to publish content. Contributing writing is terrific; there is no need to educate team members – even repeat contributors – on the publishing process.

Publishing an article: (Maybe) More than you think

When I have a final draft, I need to follow these steps before I prepare a piece of content:

1. Taking the content from a Google Document (our collaborative editing tool of choice for blog posts) and copying it into our CMS;
2. Making sure every heading has the right level (i.e. a title, subtitle, h1, h2);
3. Removing unnecessary line breaks introduced when copying the document into our CMS;
4. Providing instructions on how to create an authorship profile in our CMS, and;
5. Adding meta information (i.e. a meta description, the post URL).

It is a lot of steps, right? I don't want team members to have to worry about the steps above. I regularly have to make small changes to make sure content looks the way it should on our website (i.e. changing heading levels). I'm thrilled when people write content and I want to encourage people to contribute more. Providing the above instructions would be counterproductive to that goal.

Notably, there are idiosyncrasies with our CMS that make it unintuitive for first time users. The details can be subtle, such as the requirement that we add a line break after all callout boxes to make sure content is spaced out. Contributors don't need to worry about this. It is my responsibility as a technical writer and editor to ensure the work goes live.

Hence, I publish content in our CMS for content I have edited, or communicate with another member of our publishing team (i.e. our marketing lead, who has been through the above process many times) to help ensure a post is published.

Authorship

When contributors write blog posts, we offer to publish it under their name on our blog. This involves creating an author profile in our CMS. An author profile features the name of the contributor, a profile photo (optional, but preferred), and a bio.

During the publishing process for a contributor's first post, I tell the contributor to look out for an email invitation to our publishing tool. This ensures the contributor is expecting the invitation and knows to action on it. Then, I ask them to fill out their profile with the aforementioned pieces of information. I provide guidance on what a bio should look like, with reference to examples.

Here are some examples, with a new example on each line:

James is a Technical Marketer at Roboflow, working toward democratizing access to computer vision.

Marketing at Roboflow! My favorite thing to do is make ideas come true.

By providing examples, a new contributor can gauge what kind of information they can and may want to include in a bio.

Our CMS requires an author profile to be set up before we can publish a blog post, so I proactively communicate with contributors to make sure they set up their profile quickly.

Publishing the post

A post is in your CMS, now you can publish it. Exciting! But, don't hit publish and be done. You need to communicate to the author that their post is live. I recommend doing this as soon as you publish a post. Tell your team member that their blog post or documentation is live. Send a URL so they can see it. A published blog post or piece of documentation with their name on it can be a personally significant moment for someone.

When you have informed the author that their post is live, offer to share the post with the rest of the team. If the author wants to share their post internally first, make sure they have the opportunity to do so. When a post has been shared, make sure that all relevant parties (i.e. a sales team) sees the post.

In cases where a piece of content is particularly important to a team – for example, if a blog post highlights a use case that a sales team has seen come up in many recent leads –you may want to hand-write a message that summarizes a post and why it may be useful for another team to read. This can go a long way to ensuring that your team knows new content is available.

You should also work with the contributor on a promotion plan if content in question is a blog post that is viable to promote.

Does the contributor want to share it on LinkedIn? If so, offer to re-share from your company account. Can you set up a scheduled post in your company's social media profile to announce the content?

When new internal contributors produce content, I explicitly state in an open Slack message in our `#content` channel that it was someone's first blog post for our company. I do this because I want to make sure new contributors not only get recognition for having written a post, but also that this was their first time writing.

Lastly, make sure the contributor is aware that they can contribute more content. If there is interest but the team member doesn't know what to write, offer to share ideas that they can think about. Be clear that you are always available if and when they want to contribute again.

Reviewing the Wolfram Language Documentation

Consider the `InputField` page⁴² in the Wolfram Language documentation. This page describes `InputField`, part of the language. Above the fold, there is a big box that shows all of the ways in which you can use `InputField` with its supported arguments. Each syntax variation has a clear definition. Skimming the top of this page, I can already see six ways I could use the `InputField` method.

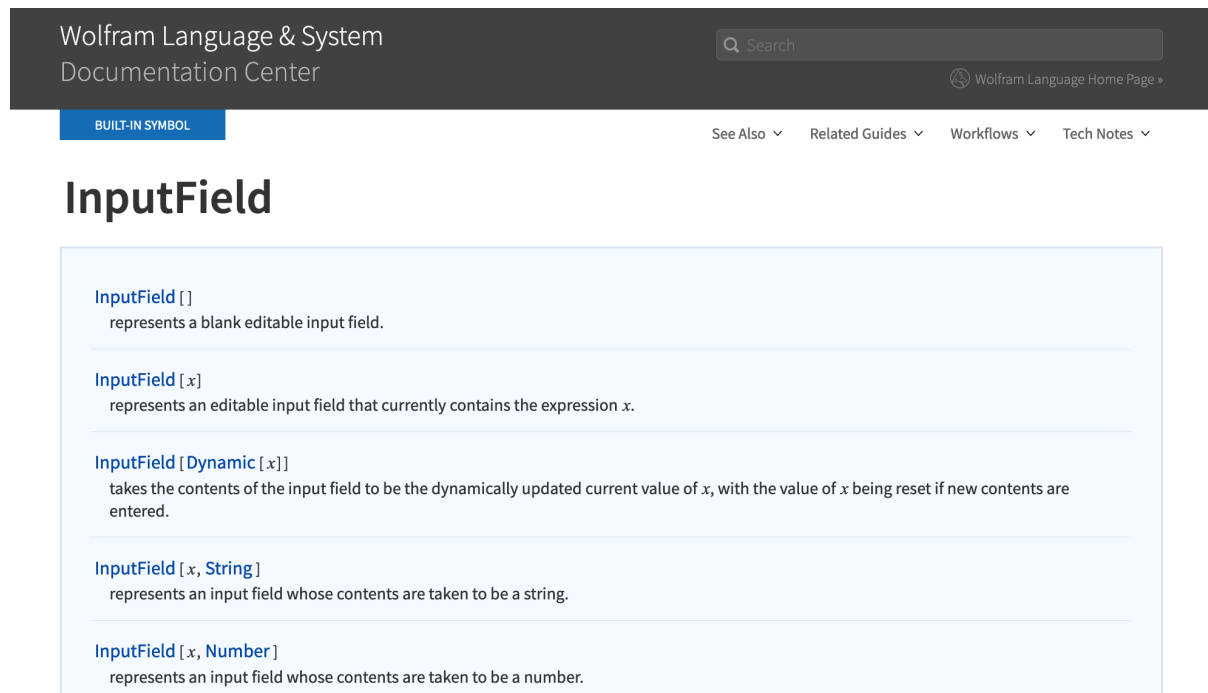


Figure 5: Wolfram Usage Examples

Scrolling down further, I can seek more information. The “Details and Options” section contains easy-to-skim information about using `InputField`. The bullet point presentation makes each point stand out. This structure makes sense because every point is a distinct piece of information about the `InputField` function.

⁴²<https://reference.wolfram.com/language/ref/InputField.html>

▼ Details and Options

- The following are possible types:

Boxes	raw boxes
Expression	expression (default)
Hold[Expression]	expression in held form
Number	number
String	string

- The setting for the input field is not updated until its contents are explicitly entered, typically by pressing `return` or by moving focus away from the input field.
- If the data given in the input field cannot be converted to the type specified, then the setting for the input field will not be updated.
- For [String](#) and [Boxes](#) types the conversion can always be done.
- For expressions, a blank input field is taken to have value `Null`. For strings and boxes, it is taken to have value `""`.
- `tab` moves between input fields.

Figure 6: Wolfram Details Examples

Then, I can see usage examples. Whereas the start of the page featured the language reference with *what* you can do with `InputField`, the examples section shows the function in action. The page succinctly shows examples of an input and the corresponding output from each example. There are more reference examples, but these are conveniently hidden behind accordions with the names of each documented attribute.

▼ Basic Examples (2)

Dynamically update the variable via the `InputField`:

In[1]:= `{InputField[Dynamic[x]], Dynamic[x]}`

Out[1]= `{x, x}`

Restrict the input to a specific type:

In[1]:= `InputField[1, Number]`

Out[1]= `1`

In[2]:= `InputField["a", String]`

Out[2]= `a`

Figure 7: Wolfram Usage Examples

Reading over the page, it is clear the Wolfram team have thought a lot about what is necessary and highlighted that information prominently. Looking at this page as a novice to the language, I feel like the page is easy to navigate. I don't feel intimidated by information overload. Specialist options – which may need to be referred to by

someone, but not most users – is hidden in accordions. In my usage of the wiki, I have seldom expanded the accordions.

Then, the page ends with links to more relevant pages. These pages link out to other language reference pages and useful guides. These features are a great way to explore the language and learn about new features. For example, when I was learning about the natural language processing capabilities in Wolfram Language I spent a bit of time clicking around to see what was possible. Wolfram made such exploration easy.



See Also

[Dynamic](#) ▪ [PopupMenu](#) ▪ [Control](#) ▪ [TableView](#) ▪ [Deploy](#) ▪ [Input](#) ▪ [Dialog](#) ▪ [FileNameSetter](#) ▪ [Placeholder](#)



Tech Notes

- [Introduction to Dynamic](#)
- [Introduction to Control Objects](#)
- [Generalized Input](#)



Related Guides

- [Control Objects](#)
- [Creating Inspectors](#)
- [Standalone Interfaces](#)
- [Creating Form Interfaces & Apps](#)
- [Custom Interface Construction](#)
- [Toolbars](#)



Related Workflows

- [Build a Manipulate](#)



History

[Introduced in 2007 \(6.0\)](#) | [Updated in 2010 \(8.0\)](#) ▪ [2016 \(11.0\)](#)

Figure 8: Wolfram Related

At the top of the page, there is a search bar with which I can find information. Of note, the “Wolfram Language Documentation” search results feature direct, one-sentence descriptions of the contents of each resource. This makes it easy to evaluate whether a page is relevant to my query without having to click through to a new page.

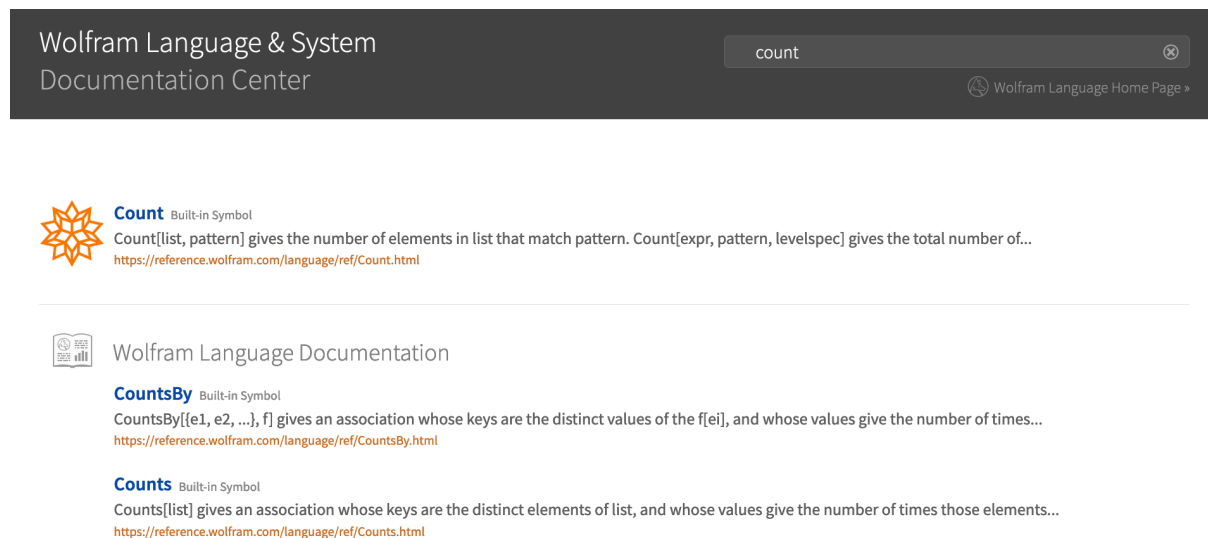


Figure 9: Wolfram Search Page example, showing the query “count” with results to relevant documentation

Overall, Wolfram’s documentation is easy to read, well-linked (which enables greater exploration), and feels more approachable than many documentation sites I have visited. Their choice to hide some information behind an accordion is inspiring. I love the pattern. Technical documentation should be written with common use cases in mind. Documenting every detail in a library or programming language is useful, but advanced options that are unlikely to be used by most users should be less prominent on the page. This is the case with the Wolfram Language documentation.

Reviewing Digital Ocean's Documentation

Digital Ocean's systems tutorials walk through everything from configuring SSH keys to setting up a web server using a tool like Apache 2 or NGINX. I found some of their tutorials immeasurably helpful to me when I was starting to learn about technology. Back when I reviewed some of their guides, I knew little about the web stack. I was exploring with curiosity in my spare time. Digital Ocean's documentation helped me learn about the web stack and, in the process, build more confidence in the command line and with Linux.

I want to dive deeper into the *how* behind the experience that I had with Digital Ocean's experience. What features and attributes does their documentation have that make it so notable? In this article, I will explore that question with reference to examples from a guide entitled "How To Install Nginx on Ubuntu 20.04⁴³". I recommend you keep the article open to the side while you are reading this blog post, if possible, as I will refer to features without direct quotes in the interests of brevity. Without further ado, let's get started!

Setting the reader on the right path: A version selector

When you work with operating systems, the area to which many of Digital Ocean's tutorials relate, versions matter. Instructions may vary between versions. Digital Ocean is not only conscious of this, but added a feature to their documentation with the importance of reading documentation for the right version in mind: you can select from a dropdown to find tutorials for different operating systems.

Here is what that feature looks like:

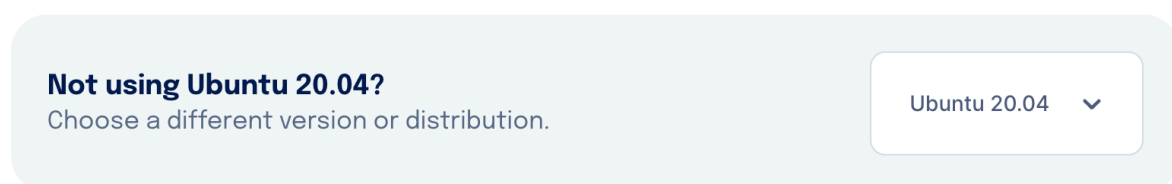


Figure 10: Choosing an operating system version

I can click a dropdown to select a new version. When I do, I am redirected to the appropriate article. This is an excellent idea. I learn both what other versions Digital Ocean has documented and I can efficiently navigate to the one I need if I have landed

⁴³<https://www.digitalocean.com/community/tutorials/how-to-install-nginx-on-ubuntu-20-04>

on a page with the wrong version. If the version of operating system I am using is not on the list, I know that Digital Ocean has not published a guide on the topic.

The introduction

Digital Ocean’s content starts with concise introductions. In two paragraphs, the author introduced:

- What NGINX is;
- Why NGINX is useful, and;
- What we will accomplish in the guide.

From reading those two paragraphs, I can evaluate whether an article is likely to solve my problem.

After the introduction, the guide notes prerequisites in a section titled “Prerequisites”. I like how this section is clearly distinguished from the introduction by using a heading. The prerequisites are direct, which removes room for ambiguity, and reference helpful resources where appropriate. Here is an excerpt from the prerequisites in the Nginx guide we are reviewing in this post:

Before you begin this guide, you should have a regular, non-root user with sudo privileges configured on your server. You can learn how to configure a regular user account by following our Initial server setup guide for Ubuntu 20.04^a.

^a<https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04>

The language is unambiguous, clear, and makes reference to a guide where I can learn more about how to meet the prerequisites.

Then, the tutorial begins.

Explaining the “whys”

This particular tutorial effectively explains the “whys” behind each command. This is particularly important for systems documentation since administrators need to know exactly what a command does before running it. Such knowledge can help them evaluate if they should run a command in the first place. Here is an example the “why” behind a command being clearly explained:

Since this is our first interaction with the `apt` packaging system in this session, we will update our local package index so that we have access to the most recent

package listings. Afterwards, we can install `nginx`:

The explanation is concise.

Copyable code snippets

You can copy-paste all of Digital Ocean's code snippets using a "Copy" button that is clearly visible in every snippet. This makes working with the documentation more efficient. You don't need to manually copy each snippet.

Snippets with outputs do not have "Copy" buttons. This small detail reduces the chances someone copies an output into their terminal, which could cause problems because multi-line outputs could break into different lines. A beginner to the terminal may struggle to figure out how to get out of a situation where they pasted multiple lines of text in a terminal. Hiding the "Copy" button from output helps to prevent this situation entirely, and reduces the chance that any developer will copy text from the guide into their terminal they don't want.

Highlighted variables to substitute

When you are writing code snippets in documentation, there are some times when you need to include placeholder values. For example, a reader may need to substitute a variable with a file name, a directory path, or another value. Digital Ocean has a feature in their code snippets that visually highlights values that need substituted.

Here is an example:



Figure 11: An example code snippet with highlighted elements where text should be changed by a user

Text that needs substituted has a different background color. This makes values that need to be changed stand out. With that said, the text highlighting is not the only cue. A reader can depend on the “your_example”, etc. text, which is important as partially sighted readers or readers with a color blindness may not notice the background color changes.

Multi-step code examples are well written

Some commands in the article have two or three steps. For example, the article describes how to enable a firewall using `ufw`, the command to verify the firewall status, and the output from the verification. The author uses short transition sentences that effectively describe what each command does. The author also specifies what the output means, where relevant.

Here is the `ufw` example, except with the output removed for brevity:

You can enable this by typing:

```
sudo ufw allow 'Nginx HTTP'
```

You can verify the change by typing:

```
sudo ufw status
```

The output will indicated [sic] which HTTP traffic is allowed:

[output]

The author described two commands and an output in three sentences. The sentences are well connected and relevant to the subject of the section, which is “Adjusting the Firewall” according to the section title that corresponds with the code snippets.

Helping readers build a solid foundation of knowledge

Going a bit further than required to help a reader build foundational knowledge, or reach an “aha moment” where they feel confident with the content in a guide is a good idea.

The Digital Ocean Nginx guide includes a section called “Getting Familiar with Important Nginx Files and Directories” before the conclusion. This section outlines some key information that you can use when working with Nginx. While the information is not technically required to install Nginx (the stated goal of the guide), it does provide the reader additional context that they can use to build confidence with Nginx.

Referring to next steps

In the Digital Ocean Nginx guide conclusion, the author summarizes the article in a sentence, then links to other articles that would be effective reading:

Now that you have your web server installed, you have many options for the type of content to serve and the technologies you want to use to create a richer experience.

If you’d like to build out a more complete application stack, check out the article *How To Install Linux, Nginx, MySQL, PHP (LEMP stack) on Ubuntu 20.04*^a.

In order to set up HTTPS for your domain name with a free SSL certificate using *Let’s Encrypt*, you should move on to *How To Secure Nginx with Let’s Encrypt on Ubuntu 20.04*^b.

^a<https://www.digitalocean.com/community/tutorials/how-to-install-linux-nginx-mysql-php-lemp-stack-on-ubuntu-20-04>

^b<https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-on-ubuntu-20-04>

As someone who has set up many web servers before (now!), I know that I can use Let’s Encrypt to set up an SSL certificate. But back when I started reading these guides, that is the kind of knowledge I didn’t have. Here, the author introduces that you can use a tool called Let’s Encrypt to add HTTPS to your domain and links to a guide you can follow. Use of the word “free” is appropriate here too since some sites charge for SSL certificates; a beginner may not know that you can get free certificates.

My conclusion

I have a bit of a soft spot for Digital Ocean's documentation because it is among the first technical documentation that I used when I was learning about basic systems administration. With that said, every time I have gone to their site I have been able to find material relevant to solving a problem. I have spent many hours learning with help from Digital Ocean's documentation.

There are some parts of their site I find a bit frustrating, like the links to "Popular Topics" in the sidebar and the amount of space that the page banner, navigation bar, and secondary navigation bar take up. But, the most important part – the content – is consistently well structured and well written.

Being a Technical Writer

Technical writers should be, as I have discussed throughout this series, integrated throughout an organization. You may work with engineers, designers, product managers, a sales team, marketing, customers, and users throughout your journey.

Each member of your team has something they can offer to help you do your job. Engineers can help communicate the “hows” and “whys” behind technical decisions and ensure your documentation is comprehensive. Designers can help you with visual assets. Product managers, marketing, and sales can all assist with positioning.

Ultimately, technical writing is all about helping people learn. You will help people use software with best practices in mind, integrate your tools into their workflows, solve problems, debug issues, and learn new skills. Thinking about the potential to help people is one of my primary motivators for being a technical writer (alongside, admittedly, my penchant for detail when it comes to documentation!).

A lot of a technical writer’s time is spent writing, but there are other tasks that are equally important. Using products yourself. Getting to know best practices. Assisting with other team members. Perhaps you have additional marketing responsibilities, too. This may especially be the case at startups, such as the organization for whom I work (Roboflow). Notably, see yourself not just as someone who documents products, but also as an advocate for customers. Use what you learn in your role to suggest opportunities for improvement. How can SDKs be improved? How can you improve workflows to encourage people to write more documentation?

Empower others to write documentation, too. Encourage your team. Documentation doesn’t have to be intimidating. A few bullet points or paragraphs for new features written by engineers – or even a screencast – will go a long way, for example. While accuracy is crucial, there is no such thing as perfect documentation (and yes, just because you wrote it, that doesn’t mean it is perfect and not open for improvement!). Prioritize accuracy and comprehensiveness. Settle for good enough. Provide feedback to your team to help them become better writers. Edit work as necessary. Where possible, make sure your work is edited, too.

There is one tiny detail that I have wanted to mention throughout this series but have never had a place: use exclamation points sparingly. I like to use one in an introduction in blog posts; the enthusiasm matches my style. In product documentation and code documentation, I avoid them entirely. Let your writing speak for itself.

Remember the moments when people thank you for your documentation, no matter how common or far apart those moments are.

Being a technical writer may sound boring, but you have an important and specialist role: you communicate complex topics in language that a target audience can under-

stand. Your work helps people use software, code, products, standards, and so many other things. As you go along, make sure you have time for fun activities. (For me, this is trying to sneak in the odd Taylor Swift reference in an exclusively marketing blog post. Yes, I'm a Swiftie. It's on my homepage, and my Spotify Wrapped 2023.)

Exercise: Review a Technical Article

As you read others' works more, you can start to build your critical eye. This eye will go a long way in helping you learn why certain properties in documentation are effective. You can use your evaluation skills to identify ways you can make your work more readable. You can also use your evaluation skills to provide feedback on technical writing produced by other members of your team.

To kickstart your building a critical thinking muscle when reading documentation, I have an exercise for you: review a technical article of mine, critically.

Below I have copied, verbatim, a technical article I wrote. The article defines image classification, various use cases for image classification, and how you can train an image classification model using Roboflow, a computer vision platform. The audience for this post is someone who wants to develop a firm understanding of what image classification is. The reader should not need any knowledge of computer vision, the field under which image classification is classified, to understand the article.

As you read, take notes on what you like and don't like about the article. Here are some questions that you can think about as you read the article:

- Are there sentences you think could be simpler?
- Are there places where the writing is not clear?
- Am I using jargon that you think is confusing
- Is there information missing that could be useful?
- What writing techniques do you like?
- Is the post appropriate for the target audience?
- Are the captions descriptive?

I encourage you to take notes as you go.

After reviewing the article below, I encourage you to find an article or piece of documentation relevant to your day-to-day work or interests that you can review. The more you practice reviewing documentation, the stronger your evaluation skills will become.

Imagine that you are working for a construction site. You have a camera set up that triggers when someone drives a vehicle onto the site. Your job is to have the camera take a photo and assign a label to the vehicle that comes in. This is an important role: the site has limited space and having too many vehicles on site can be a safety hazard.

How can we solve this problem? This is where image classification comes into play.

In this article, we are going to discuss:

1. What is single-label and multi-label image classification?

2. In what situations is image classification useful?
3. How does image classification compare to other computer vision algorithms?
4. What real world use cases exist for image classification?
5. How you can classify images with Roboflow.

Let's get started.

What is Image Classification?

Image classification is a computer vision task where label(s) are assigned to an entire image. The label should be representative of the main contents of the image. For instance, you could have a classifier that identifies whether a bird is, or is not, a certain species or sex.

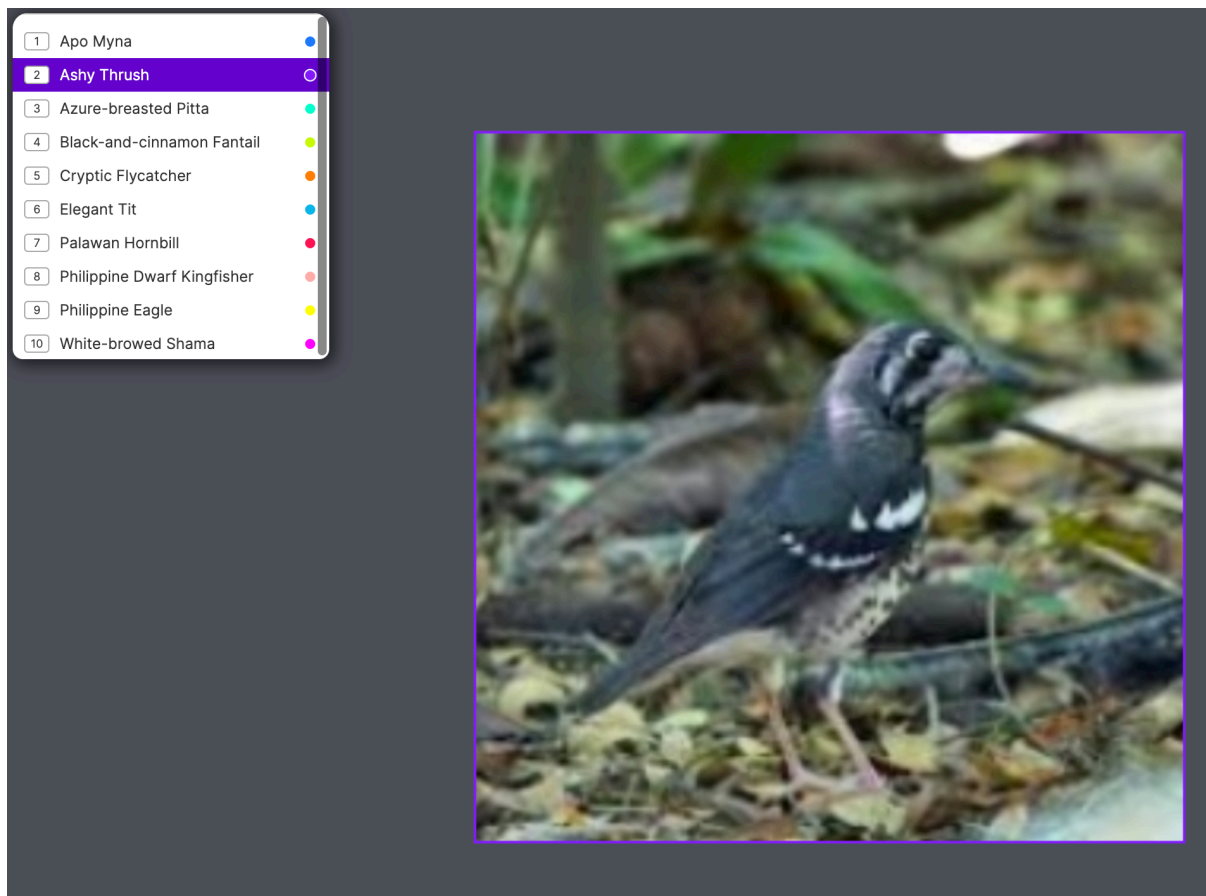


Figure 12: img

Image classification models are useful when you need to answer a question in this form:

“In what category or categories should the image be placed?”

An image classification algorithm will present multiple options alongside confidence levels. These confidence levels refer to how sure the classifier is that an image falls into a category. In production, you would take the classification with the highest confidence level and use that information to do something else (i.e. re-route an object to a different conveyor belt).

For instance, a coffee plant species detector might return a result like this:

1. Coffea Arabica (95.001%)
2. Coffea Robusta (4.899%)
3. Coffea Eugenoides (0.1%)

In this case, the detector has returned three possibilities. Coffea Arabica has by far the highest confidence level. As long as the algorithm has been trained well and is working as expected, we could confidently say that the image contains Arabica coffee.

There are two types of image classification algorithm:

1. Single-label classification
2. Multi-label classification

Let's talk about each of them.

What is Single-Label Image Classification?

Single-label image classification algorithms assign a single, specific label to an image. A single-label classifier could say that an image contains a dog or a cat, for example. Single-label classification is appropriate when you only need to detect one type of object in an image (i.e. whether an image contains a forklift, a car, or a truck).

Let's go back to our earlier scenario where we want to determine the species of coffee plant species. This is a problem that can be solved with single-label classification. In this scenario, we only need to know the species of a coffee plant. This only requires use of a single label. The label might say “Coffea Arabica”, for example, if the coffee is of the Arabica varietal. Coffee plants cannot be two types at once, so single-label classification works well.

What is Multi-Label Image Classification?

Multi-label classification algorithms assign multiple labels to describe the contents of a whole image. For example, a multi-label classifier could say that an image contains

both a forklift and people. But, the classifier cannot point out exactly where the objects are in the image.

Multi-label image classification is useful when there is more than one feature in a single image that you want to identify. Imagine you are working to implement a plant disease classification algorithm. Your algorithm needs to be able to identify both the plant type – tree or sapling, in this example – as well as whether the plant is showing symptoms of disease. This is a task suitable for multi-label image classification.

Imagine we have an image of a diseased sapling. The multi-label classifier could return that an image contains two labels: sapling and diseased. A single-label classifier, however, could only tell us whether the plant was a sapling or diseased, unless we specifically trained the algorithm to recognize diseased saplings. This would be sub-optimal because disease symptoms would overlap since saplings are young trees.

Image Classification Use Cases

When might you want to use image classification? That's a great question.

If you are looking to assign a single category to the contents of an image, a classification algorithm is a good decision.

In addition, if there are a limited set of categories in which an image can be classified, a classification algorithm is appropriate.

Consider the following scenario. You need to identify whether a banana is ripe, not ripe, or over ripe⁴⁴. Bananas are considered unripe if they are green, and are ripe if they are yellow. If a banana is starting to turn brown or black, the banana is considered over ripe. To humans, this identification is relatively easy. We can look at the banana and can usually make a determination.

In this scenario, an image classification algorithm would be helpful. You could set up a camera to take photos of bananas as they come in on the production line. If a banana is green, it is not yet bundled for sale in a supermarket. If a banana is yellow, it is prepared for shipment. If a banana is overripe, the banana is separated to be used in compost.

With this system, you can ensure:

1. Your business doesn't distribute too many bananas that are not ripe enough, which might impact day-to-day sales for some supermarket customers.
2. Ripe bananas are sent to supermarkets which customers prefer.
3. Overripe bananas are put to good use, reducing wastage if those bananas were to get to customers who don't have a compost system set up.

⁴⁴<https://universe.roboflow.com/bottle-wl3fj/banana-detection-wzs7e?ref=blog.roboflow.com>

We have just solved a real-world business problem with image classification.

How else might image classification be used? Let's talk through a few more examples:

- **Defect detection:** An image classifier could detect whether or not materials produced on a production line are or are not defective. Specific defects could be made part of the classifier (i.e. the product contains a hole, the product is cracked).
- **Not safe for work (NSFW) content moderation:** An image classifier could ensure NSFW content cannot be posted on a platform that does not allow such content.
- **Leak detection:** With an image classifier, you could monitor pipes throughout a factory and flag any instances where pipes are leaking.

These are a few of the many real-world applications of image classification. Remember, if you need to decide on whether the contents of an image fall into one of multiple categories, image classification can help.

How to Classify Images with Roboflow

Using Roboflow, you can classify images⁴⁵. First, create an account on the Roboflow website⁴⁶. Then, create a new project in the Roboflow dashboard.

For our example, let's create a project that classifies sports from an image. Specifically, we want to find a label that describes whether an image contains a game of cricket, baseball, or football. In business, this could be used to build a database that aids in searching images and videos. If the sport in an image is in our list of sports we can identify, we should be given a classification; otherwise, we should get no result. Let's call our project "Sports Classifier".

Select the "Single-label image classification" label in the "Project Type" dropdown. Here's how your project should look:

⁴⁵<https://blog.roboflow.com/label-classification/>

⁴⁶<https://app.roboflow.com/?ref=blog.roboflow.com>

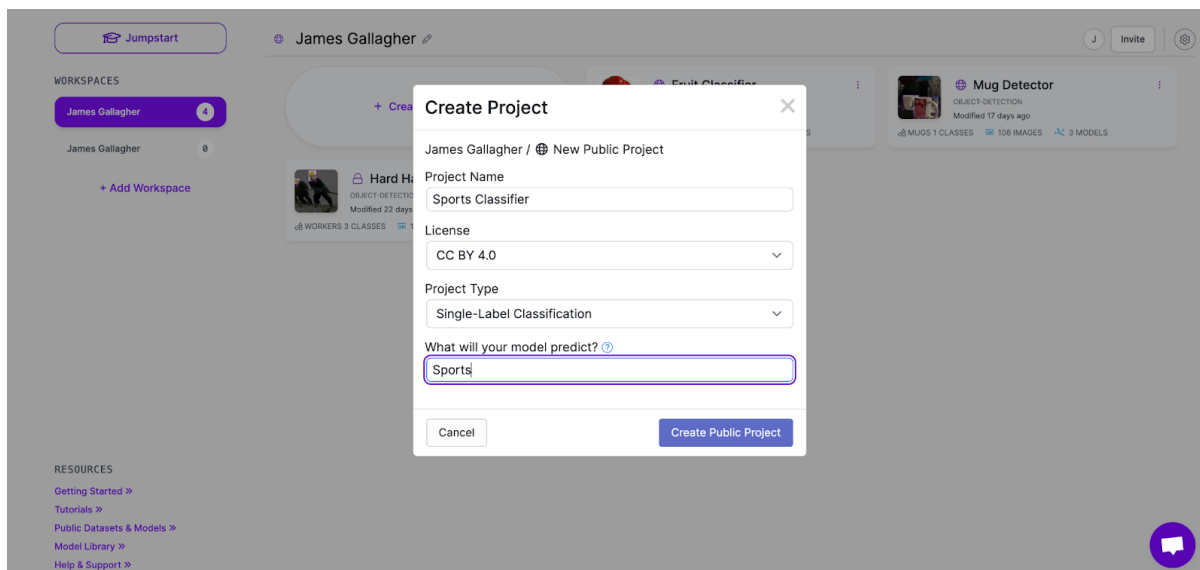


Figure 13: img

Let's start by copying a sports classification dataset from Roboflow Universe⁴⁷, our repository of more than 100,000 open source datasets:

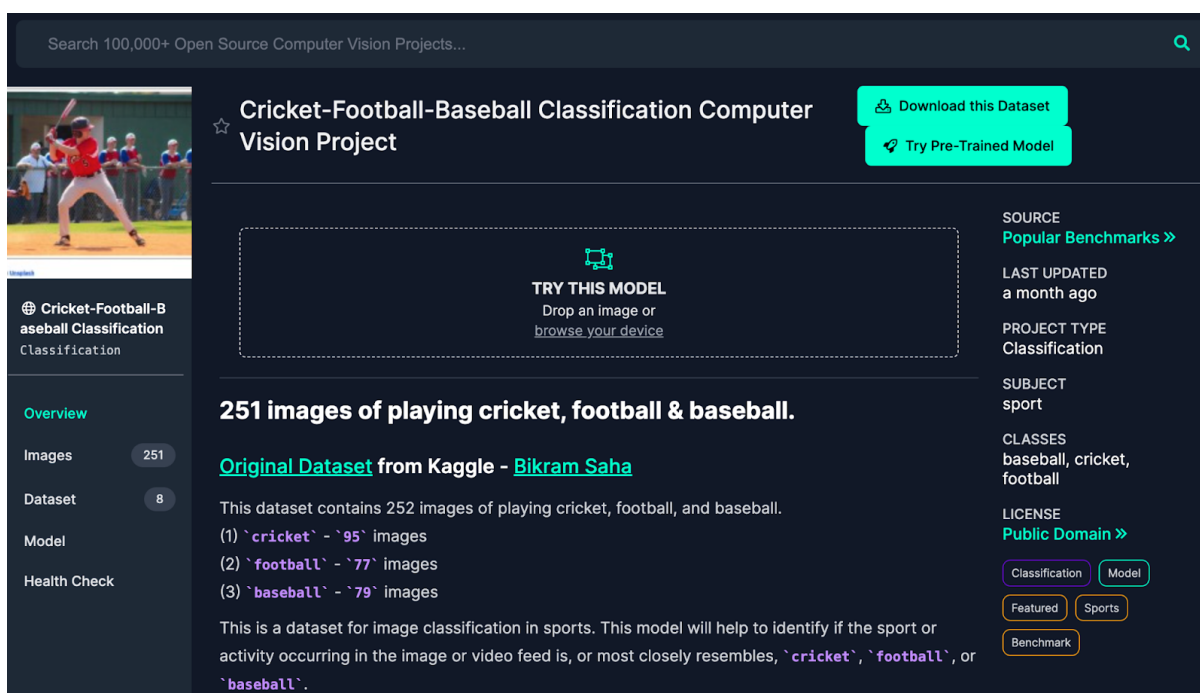


Figure 14: img

Next, let's download the dataset and upload it into our new project in the Roboflow application⁴⁸:

⁴⁷<https://universe.roboflow.com/?ref=blog.roboflow.com>

⁴⁸<https://app.roboflow.com/?ref=blog.roboflow.com>

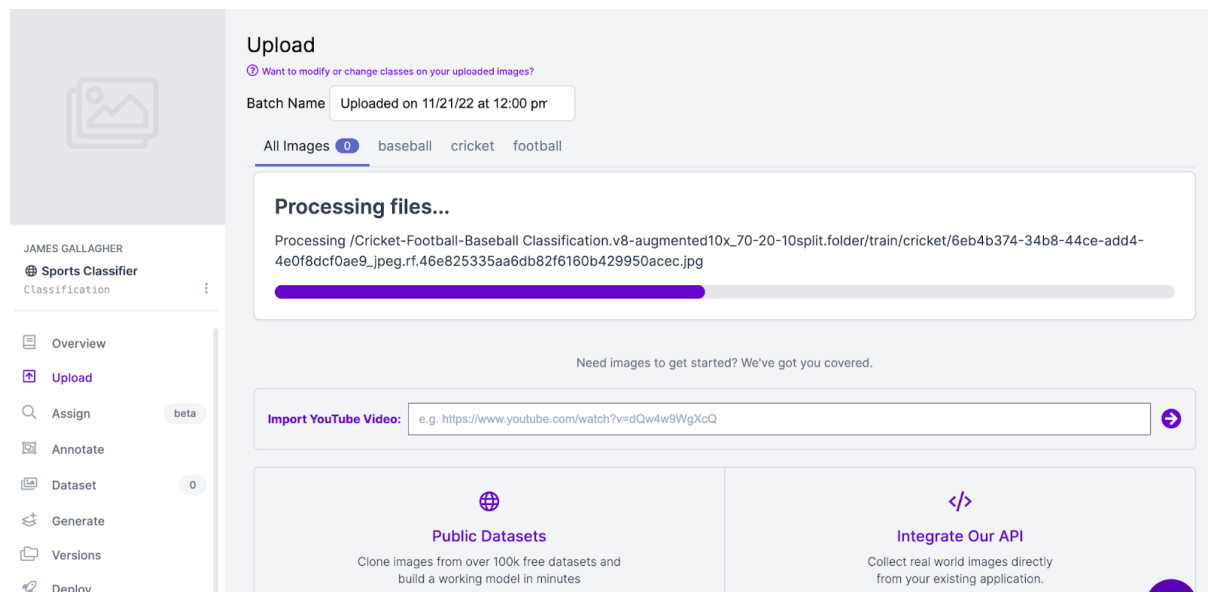


Figure 15: img

We now have the data we need to perform our image classification. But, we don't have a model yet. That's our next step. To start classifying images, we need to train a model. To prepare to train our model, we need to:

1. Click "Save and continue" once all of our images are in Roboflow in the page where we uploaded our images above.
2. Select a train test split from the pop up. You should click on the Method form and then click "Split images between train/valid/test" to make sure your images are split properly. 70% of images will go in your training set, 20% go in your validation set, and 10% go in your testing set.
3. Click "Approve All" and "Add Approved to Dataset" once all of your images are uploaded to add your images to your model. Then, click "Add images".
4. Click the "Generate a new version" button.
5. Now, we're ready to train our model.

The "Generate a new version" button takes you to a page that lets you customize your new version of your model. For now, let's leave all values the same as they are. We can click "Generate" in the final tab on the page.

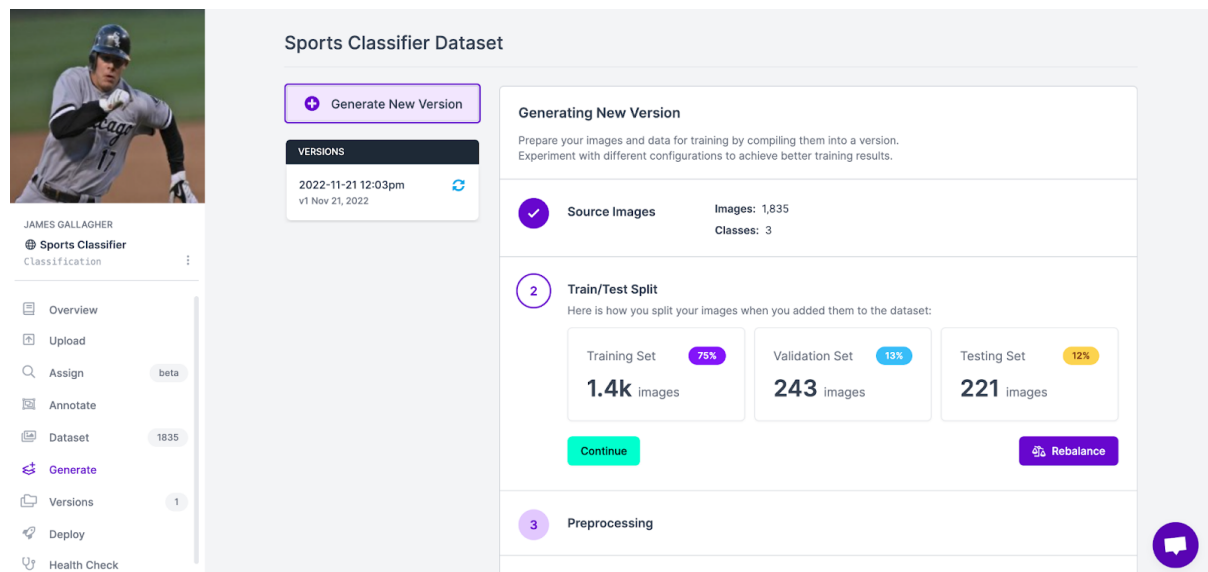


Figure 16: img

Next, we need to click “Train” on the training dashboard so that Roboflow knows you are ready to train the version of the model you just created.

Now, we need to wait for our model to train. This can take anywhere from a few minutes to a few hours, depending on how large your dataset is. Make a cup of your caffeinated beverage of choice and come back to the Roboflow dashboard when you get an email notifying you that your project has been trained.

Once you have a trained model, you can start to use it. Click the “Deploy” tab, where you will see many different ways in which you can deploy your model. For this example, we’re going to upload an image from the internet and see how it is classified. Let’s upload an image of a few people playing cricket and see what happens:

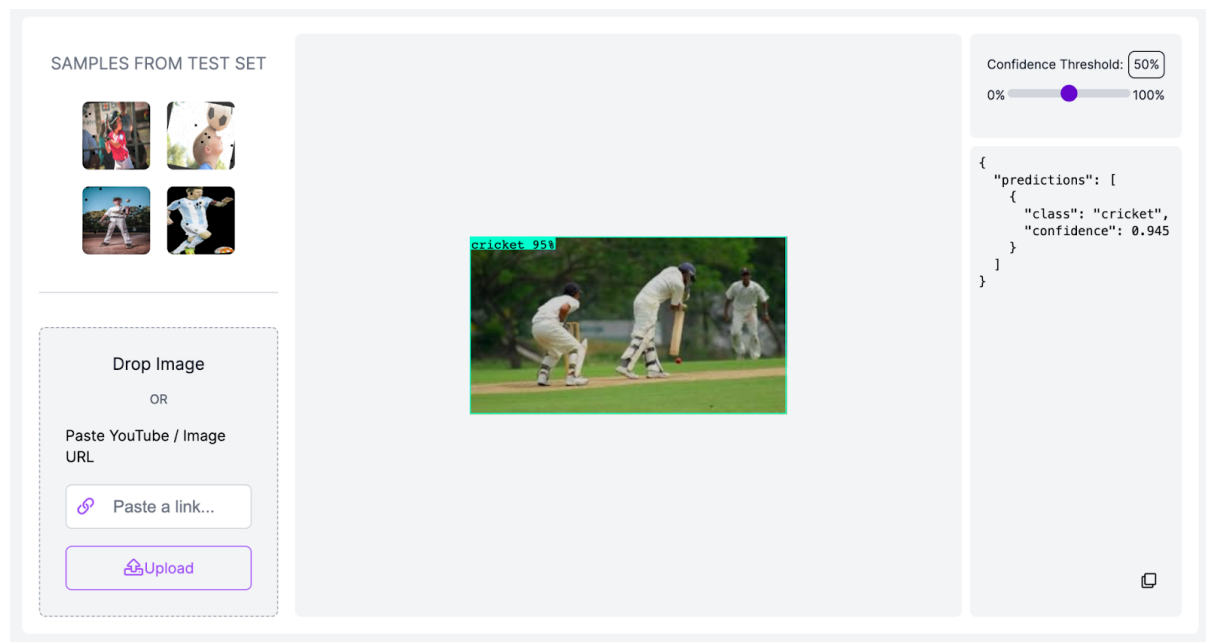


Figure 17: img

Our image has been given the label “cricket” with a confidence rate of 95%. Our image classification model is working!

Using this same approach, you can generate classification models for a vast range of different problems, from detecting defects in a production pipeline to classifying animals that are found on a farm.

Now you have the skills you need to generate an image classification model. If data does not already exist for your problem, however, you will need to annotate images yourself. To find out more about how to do this, check out our image annotation tutorial⁴⁹.

Image Classification Compared to Other Algorithms

Image classification is only one of many key algorithms used in computer vision. Indeed, while image classification is immensely useful and practical for some problems, in other instances there are more suitable algorithms to use. Let’s discuss how image classification compares to other algorithms.

⁴⁹<https://www.youtube.com/watch?v=kr3rvqWLEFE&ref=blog.roboflow.com>

Image Classification vs. Object Detection

Both image classification and object detection algorithms⁵⁰ assign labels based on the contents of an image. However, image classification labels a whole image, whereas object detection algorithms can label specific instances of objects in an image.

Imagine you need to find all of the boxes in an image to make sure there are not too many on a construction pallet⁵¹. Image classification could only tell you boxes are on the pallet. That is not helpful information in this context. Object detection, however, could tell you where all of the boxes are. You can count each instance of a box to see how many boxes there are in the view of the camera.

Image Classification vs. Instance Segmentation

Like image classification, instance segmentation⁵² returns a label determined by the contents of an image. Unlike image classification, segmentation algorithms add a label to the exact pixels where the instance of an object is located in an image.

Imagine you want to identify not only that a product contains a screw that has not been screwed in properly, but the exact location of the screw in the image where the defect has been highlighted. This is possible with instance segmentation. With instance segmentation, the screw that has not been screwed in properly will be highlighted so you can see the exact location of the defect in the image.

Conclusion

Image classification is useful for solving problems where you need to classify an object into one of multiple groups. This computer vision algorithm has wide uses across industries, from content moderation to defect monitoring.

With the information we have discussed thus far, you now have the information you need to explain instance segmentation, the types of problems that are best solved with instance segmentation, and how it compares with a few other computer vision algorithms.

⁵⁰<https://blog.roboflow.com/object-detection/>

⁵¹<https://universe.roboflow.com/raja-sekar-mrubn/demobox/dataset/4?ref=blog.roboflow.com>

⁵²<https://blog.roboflow.com/instance-segmentation-roboflow/>

ISBN



Figure 18: ISBN