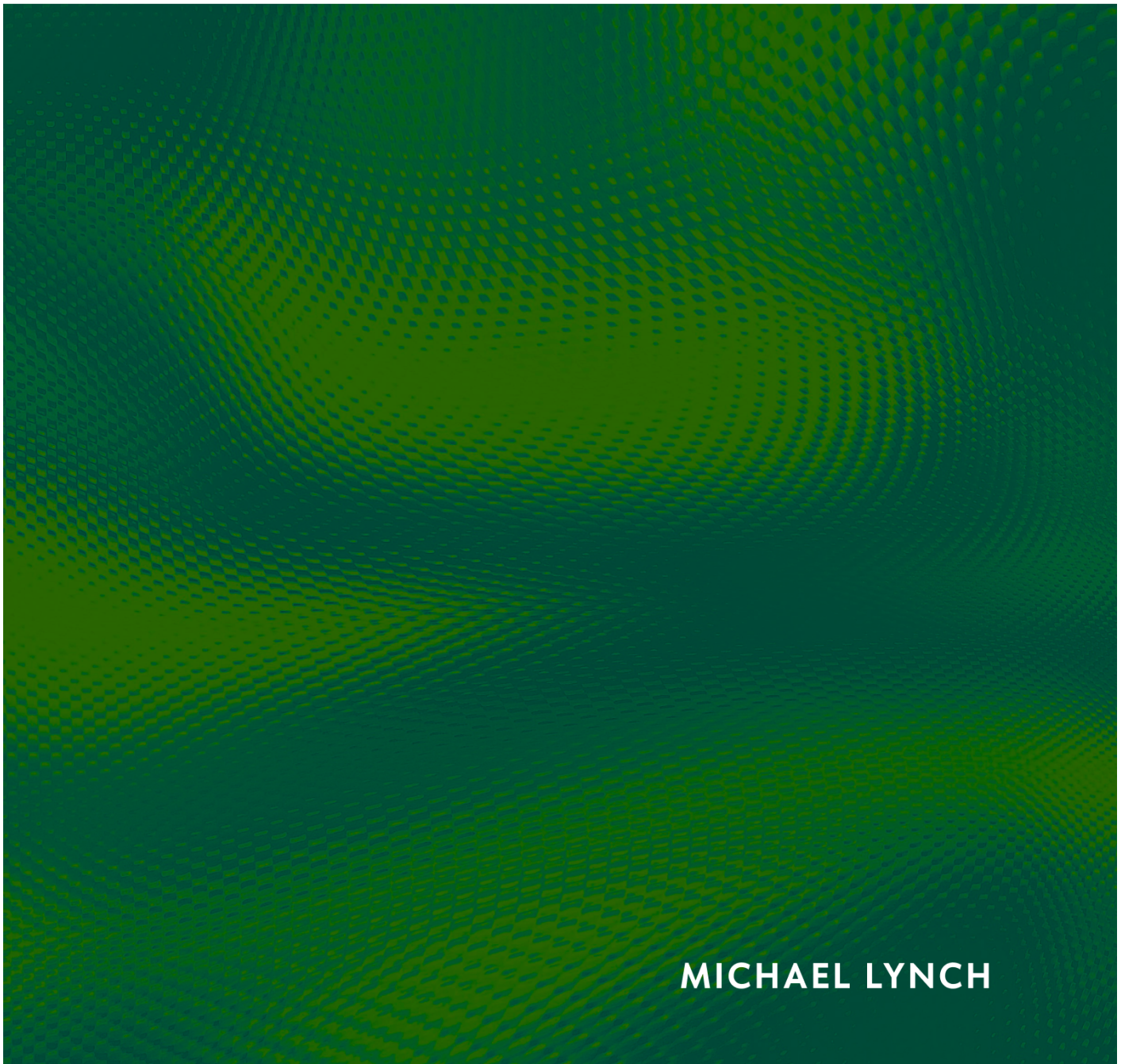
REFACTORING ENGLISH

Effective Writing for Software Developers



MICHAEL LYNCH

Refactoring English

Michael Lynch

Version v0.6.1, 2025-07-21: Pre-release

Table of Contents

Preface: Why Improve Your Writing?

Coming soon

Chapter 1. Grab the Reader's Attention

Coming soon.

See Get to the point for a preview of this chapter.

Chapter 2. Write Blog Posts that Developers Read

When I first learned to write software, I read tons of software blogs, but I didn't dare start my own. I felt like I didn't have anything smart or interesting to say.

I started my software blog in 2016, a full decade into my career as a software developer. Once I started writing, I realized that there's a near-infinite amount of interesting things to say about software, and there's no minimum smartness requirement to start a blog.

Writing a blog is one of the most rewarding things I've done in my career. It continually sharpens my skills as a developer, exposes me to new ideas, and connects me with interesting people.

Every time I write about something, it makes me better at whatever I'm writing about. Explaining my ideas forces me to challenge my assumptions and consider alternatives. Often, the act of explaining my thought process reveals a better solution than what I sat down to write about.

As my blog has grown in popularity, it's also boosted my career. When I started a new business in 2020, all of my early customers discovered my product through my blog. When I email people I've never met to ask questions or propose that we collaborate, they often tell me that they agreed because they enjoy my blog. Odds are, you discovered this book because of my blog.

In my nine years of blogging, some of my articles have reached 100k readers in a single day, while others have flopped. Through trial and error, I've discovered ways to reach more readers and present my ideas in ways that resonate with them.

Some blogging success comes down to luck, but more of it is within your control than you realize. In this chapter, I'll share the practical techniques you can use to write better blog posts and reach more readers. You'll learn:

- How to think of ideas for blog posts
- How to screen ideas based on their likelihood of reaching readers
- How to use your blog to find customers for your product or service
- How to give your readers the best possible reading experience

2.1. You're qualified to write a blog post

One of my teammates once told me that he wanted to try blogging, but he didn't know what to write about.

He was a sharp developer, and he frequently taught his colleagues new programming techniques. I suggested that he write about Bash programming, as his Bash scripts were exceptionally clear despite his only learning the language six months prior.

"Oh, I can't write about that," he said. "I'm no Bash expert."

2.1.1. You don't have to be an expert

So many developers think there's a rule saying you can't write a blog post until you're an internationally-recognized expert in the subject. It's not true.

Imagine that you'd never written a line of C++ in your life, so you spend 20 hours learning enough C++ to write a few toy programs. You write a blog post about what you learned and what resources helped you. Another C++ beginner reads your blog post and reaches your skill level in only 10 hours. That would be a useful blog post, and you'd be entirely qualified to write it.

Maybe you're thinking, "Sure, I could write a blog post that saves a beginner 10 hours, but the world's greatest C++ developer could probably write a blog post that saves the reader 15 hours."

There are two flaws with that reasoning:

1. The world's greatest C++ programmer isn't writing blog posts about the basics of the language.
2. Even if the world's greatest C++ programmer tried to write an introduction to the language, yours would be better.

Claim (1) is unsurprising. The world's greatest C++ developer is probably off earning six figures per week fixing mind-melting bugs for mega-corporations, not writing basic tutorials.

Claim (2) might surprise you, but it's true. The world's greatest C++ developer doesn't remember how to learn C++ because they did it decades ago. And even if they remember, the way they learned back then isn't the best way to learn today.

Worst of all, the world's greatest C++ developer spends all their time with other experts. The way they think about the language is totally different from how a beginner would.

There's a name for the difficulty that experts experience in relating to beginners: The Curse of Knowledge. Because they've been deep in a subject for so long, experts struggle to explain

ideas to non-experts. If you've ever asked a computer science PhD what their doctoral thesis was about and then nodded politely throughout their rambly, incomprehensible answer, you've witnessed the curse.

More often than not, a lack of expertise is an advantage in blogging, not an obstacle.

2.1.2. Don't overstep your expertise

What happens if you write about something before you're an expert, and then someone smarter reads it? Will they mock your blog post and denounce you for even trying to write about it?

The Internet is teeming with grumpy readers who consider themselves experts, but you can protect yourself from their bullying and nitpicking.

Typically, people get angry about blog posts not because the author has too little expertise but because they accidentally presented themselves as an expert. I call this "overstepping your expertise."

If you just learned web development a week ago, you'll probably get hostile responses if you write articles with titles like "Every Web Framework is Dumb" or "10 Rules that Every Professional PHP Developer Absolutely Must Know." Posturing in this way suggests a level of authority that doesn't match your experience.

Writing at your level of expertise doesn't mean apologizing to the reader or couching everything you say in self-doubt. It just means presenting what you know in an honest way rather than pretending to know more than you do.

You can avoid overstepping your expertise by following two simple guidelines:

1. Write descriptively rather than prescriptively.
    - Write what you personally found effective rather than prescribing to your readers what they should do.
2. Instead of critiquing the technology, describe your experience learning it.
    - Resist the temptation to denounce an API or language as "bad." Instead, say that you found it confusing. You're qualified to speak authoritatively about your own experience. Leave room for the possibility that the technology has constraints or design choices you don't yet appreciate.
    - Conversely, avoid saying that something is the "best" way of doing something. Without experience, you can't assess the "best" solution. Instead, present it as the best way that you've found.

Consider this explanation of the `set -u` option in Bash:

BAD: *Adopt an authoritative tone and speak in absolutes.*

All Bash scripts should begin with the `set -u` option. It's a critical feature that prevents unexpected behavior at runtime.

The above sentence adopts an authoritative tone, implying that the author is a Bash expert. It's prescriptive rather than descriptive, meaning that it tells the reader what to do rather than what the author does.

In contrast, consider this alternative wording that avoids overstepping the author's expertise:

GOOD: *Describe your personal experience.*

The `set -u` option made it so much easier for me to write Bash scripts. The default behavior in Bash is to treat undefined variables as empty strings, which made debugging harder. When you enable the `set -u` option, Bash exits with an error whenever it encounters an undefined variable. That option helped me catch bugs early, as I've never had a situation where I want Bash to treat an undefined variable as an empty string.

The second description of the `set -u` command avoids an authoritative tone and presents information descriptively.

The second version is also harder to nitpick. Even if the world's grumpiest Bash expert reads your post, what is there to argue against? No matter how much someone knows about Bash, you're still correct in claiming that you personally had a better experience after you discovered the `set -u` option. They might have some "well, actually" counterexample, but they'll be far less feisty than had you decreed that everyone needs to set this option all the time.

As you gain more comfort and experience in a technology, you should strengthen your language to convey more authority. Over time, you'll develop a more nuanced view of techniques by seeing where they work and where they don't. As you develop this more mature perspective, you can speak more authoritatively rather than presenting yourself as a beginner.

2.1.3. Two people can write about the same topic

What if you have an idea for a blog post and see that someone has already written something similar? That's okay. There's enough room on the Internet for two articles about the same thing.

In music, it's common for multiple artists to perform the exact same song. For example, the Beatles released "Lucy in the Sky with Diamonds" in 1967, and several artists have since released their own cover versions, including Elton John, The Black Crowes, and Miley Cyrus. Each performer brings their own voice and style to the song, so different people prefer different versions.

So, even if there are already articles about a topic you want to blog about, some readers will prefer the way you "sing" it. You'll naturally have a unique way of explaining concepts and choosing what details to include, and some readers will prefer your version to anything else out there.

The only time existing articles should worry you is if there are dozens of them from more established websites. In those cases, all the noise will make it hard for readers to find your post (see Plan a path to your readers).

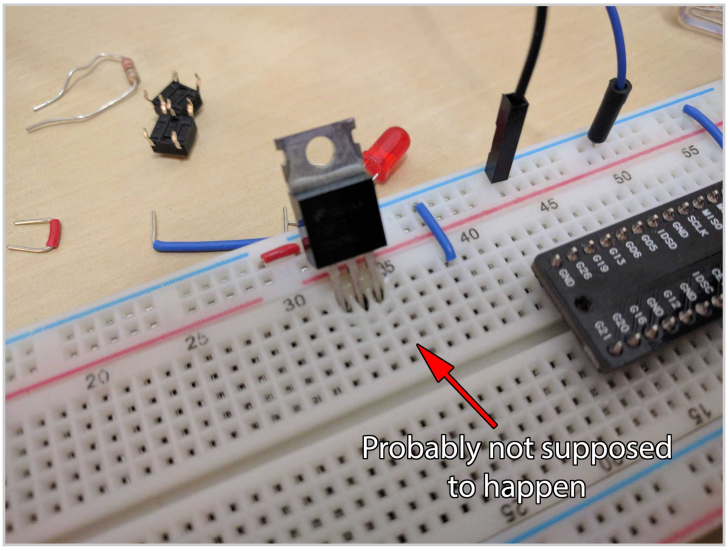
2.1.4. Case study: Leaning into my weaknesses as an electrical engineer

A few years ago, my friend and I built a robot named GreenPiThumb to automatically water my plants, so I wrote a blog post about it.

GreenPiThumb was my first time working with electronics and circuits, and I found a lot of the concepts confusing. When I wrote the first draft, I tried to pretend I knew what I was talking about.

Immediately, my writing felt phony, and it was. I was misrepresenting how much I knew. Worst of all, faking knowledge was hard. For every sentence I wrote, I had to do hours of research to make sure I wasn't saying something wrong.

Instead, I tried a different strategy. I embraced the humor of our inexperience. During the project, so many things went wrong because we didn't know what we were doing, so I wrote about that. I joked about how I wired a transistor incorrectly, and it got so hot that it melted part of our equipment.



That article became one of my first hit blog posts, reaching 37k readers in its first week. Several other bloggers had written about similar plant-watering robots, but GreenPiThumb gained more attention.

I suspect part of our relative success was that we were free from The Curse of Knowledge. The other posts assumed the reader understood hobbyist electronics, but we showed how far the average person could go with no electronics background at all.

2.1.5. Case study: Julia Evans' immunity to The Curse of Knowledge

Julia Evans is a popular software blogger, known for her tech-oriented zines. One key to her success is a seeming immunity to The Curse of Knowledge.

Julia has an astonishing ability to retain her learner's mindset long after she gains expertise in a subject. She writes with warmth and empathy, using language that's approachable regardless of the reader's prior knowledge.

One way that Julia avoids The Curse of Knowledge is by writing as she learns. Her 2023 post "Some notes on using nix" is a good example of her writing about a technology as she learned it:

> I've been trying to figure out how to use nix in a way that's as simple as possible and does not involve managing any configuration files or learning a new programming language. Here's what I've figured out so far!

You can see from the excerpt that Julia adopts a descriptive, non-authoritative tone. She makes it clear that she's not an expert, but she's sharing what she's learned.

Julia's Nix post is also non-judgmental. Even when she runs into rough edges, she never says Nix is dumb or bad, just that it doesn't work the way she expected:

> There's a more aggressive version of `nix-collect-garbage` that also deletes old versions of your profiles (so that you can't rollback)
>
> ```
> $ nix-collect-garbage -d --delete-old
> ```
>
> That doesn't delete `/nix/store/8pjnk6jr54z77jiq5g2dbx8887dnxbda-oil-0.14.0` either though and I'm not sure why.

You might think, "Why would I want to learn Nix from someone who doesn't really understand it?" But I can tell you that Julia's post got me to start using Nix. I'd been interested in the technology for a long time, but every other Nix article I'd found was filled with jargon and assumptions about my background knowledge. Julia's post was the first one that met me at my level and showed me the basics of Nix in terms that made sense to me.

2.2. Choosing topics

Developers often hesitate to start blogging because they can't think of anything to write about. They feel like everything they're qualified to explain, someone else has already explained.

In the 2002 film, *Adaptation*, a struggling screenwriter asks his instructor how to write a screenplay that's like "the real world," where "nothing much happens." The question infuriates his instructor, who spends the next two minutes eviscerating him:

> Nothing happens in the world? Are you out of your fucking mind? People are murdered every day. There's genocide, war, corruption. Every fucking day, somewhere in the world, somebody sacrifices his life to save someone else. Every fucking day, someone, somewhere takes a conscious decision to *destroy* someone else. People find love; people lose it. For Christ's sake—a child watches her mother beaten to death on the steps of a church. Someone goes hungry. Somebody else betrays his best friend for a woman.
>
> If you can't find that stuff in life, then you, my friend, don't know crap about life.
>
> -*Adaptation* (2002)

Your blog posts don't need to be as dramatic as all that, but if you feel like you're not experiencing anything worth writing about, you're probably missing things.

As a developer, you're learning new things all the time. Even if your work feels repetitive, the way you'd solve a problem today is not the same as you would a year ago. Somewhere along the line, you've learned something worth sharing.

2.2.1. How I think of blog topics

When I started my blog, I barely had topic ideas. As I write more and see what readers respond to, it's become easier to recognize when I can translate an experience or passing thought into an interesting blog post.

Most of my blog topic ideas come from the following questions:

What's a technology I want to learn more about?

Blogging is a great way to learn new technologies. By explaining what you learn as you go, it solidifies your understanding of the material and helps you think about the technology in a systematic way.

Beginner blog posts also attract expert feedback. Experts don't want to deal with lazy learners who can't put in effort of their own. When you blog about what you learn, it shows experts that you're willing to do hard work and contribute back to the community. I frequently receive feedback from experts, including the lead developers of the language or

tool I'm writing about.

Examples:

- My First Impressions of Gleam
- Using Nix to Fuzz Test a PDF Parser
- Using Zig to Call C Code: Strings

Did I earn or lose a lot of money on something?

People are generally reluctant to talk about money, so if you share details about your finances, readers get excited. The downside is that readers pay outsized attention to the financial details, even if they're a minor part of your post.

Part of the fear around financial transparency is that a competitor might take advantage of the information you share. That's a legitimate concern, but people overestimate the risk. I've been largely transparent about my business finances for seven years, and I've never seen it give a meaningful advantage to a competitor. Instead, I've gotten helpful feedback and greater visibility for sharing details of my work.

Remember that transparency isn't all-or-nothing. You choose what to disclose, you can do so strategically, and you can stop whenever you want. For example, if I found an amazingly effective website to advertise my product, I wouldn't blab to everyone about it because that's too easy for a competitor to steal.

Examples:

- I Sold TinyPilot, My First Successful Business
- I Regret My $46k Website Redesign

What did I learn from a video or podcast?

If I learned something useful from a podcast or video, I sometimes publish my notes as a blog post. I'm spending the time listening or watching anyway, so the marginal effort to publish my notes is small. Sometimes these posts attract a surprising number of readers.

I typically publish notes about podcasts and videos, but you can do this with any media that's not in a web-friendly format. For example, in 2024, Simon Willison published notes about a leaked PDF from a popular YouTube star. All he did was quote the salient points from the document and add a few sentences of light commentary. I bet it took him less than an hour to write, but it became one of Hacker News' most popular posts of the year.

Examples:

- GUIs are Antisocial
- Designing the Ideal Bootstrapped Business

My other common topic ideas

Topic Idea	Examples
What showcases a product I made?	• TinyPilot: Build a KVM Over IP for Under $100 • Building a Budget Homelab NAS Server
What did I recently explain to someone?	• if got, want: A Simple Way to Write Better Go Tests • Guidelines for Freelance Developers Working with Me
What story do I keep telling friends and teammates?	• How I Stole Your Siacoin • I Regret My $46k Website Redesign • Adventures in Outsourcing: Cooking with TaskRabbit
What's an idea I'm passionate about?	• How to Make Your Code Reviewer Fall in Love with You • Why Good Developers Write Bad Unit Tests
What did I learn from a recent project?	• Why does an extraneous build step make my Zig app 10x faster? • Stripe is Silently Recording Your Movements On its Customers' Websites

My topic strategy is not optimized for gaining subscribers

Most popular software bloggers have a narrower focus than I do. For example, John Gruber mainly writes about Apple and Josh Comeau writes about frontend development.

I bias my writing towards my own enjoyment. There's no single topic I'd enjoy writing about for years, so I flit around from topic to topic. I sacrifice blog subscribers for the sake of enjoying what I write.

When you choose topics, consider the tradeoff between building an audience around a narrow focus and exploring your curiosity with a broader focus.

2.2.2. What topics I avoid

There are some topics that I deliberately avoid either because they're not worth writing about or because their net impact on the world is negative.

I must write this even if nobody reads it

I used to have an overwhelming desire to write certain posts regardless of whether anyone would read them. I'd think, "I've wasted so much time learning this thing nobody else has documented, so I *have* to explain it. Audience be damned!"

That attitude ended when I wrote "Hiring Content Writers: A Guide for Small Businesses." It's the longest piece I've ever written, and I knew while I was writing it that I'd have a hard time getting anyone to read it.

I published the article and found no useful places to share it. The regular readers of my software blog weren't interested in a 45-minute read about hiring content writers. And search engine results for the topic were filled with spammy sites that I could never outrank. It wasn't a match for any forum, either.

After that experience, I decided that my writing time was finite, so why spend two months writing an article that nobody will read? I can use that time to write articles that will potentially reach thousands of interested readers.

It still pains me to learn something that nobody has documented and then move on without adding any documentation myself. As a compromise with myself, I write shorter, messier posts in a separate "Notes" section of my blog. My notes are deliberately less polished than my other articles, but they teach the important parts of what I learned.

Starting an argument / attacking people

Some bloggers enjoy arguing online, so they publish inflammatory posts about why some technology sucks or why PHP is better than Ruby. I admittedly enjoy reading those posts, but I don't enjoy being part of online fights. It's just too much of a time suck.

I also avoid attacking people or businesses, even if I think they're doing something unbelievably stupid. I'll make exceptions if they do something harmful, but even then, I prioritize it lower than articles where I'm just teaching something rather than stoking conflict.

For example, I criticized Stripe, the payment processing company, for what I viewed as egregious privacy violations. But that post required a lot of extra work, as readers questioned my findings, and an executive from Stripe reached out for a 1:1 conversation. I was glad to see the post make an impact, but it reminded me that critical posts come with extra baggage.

2.2.3. How I rank ideas

I'm a slow writer, so I always have a backlog of post ideas that I haven't written yet. When I'm deciding which idea to grab from my backlog, I choose based on several criteria:

- How excited am I to write this?

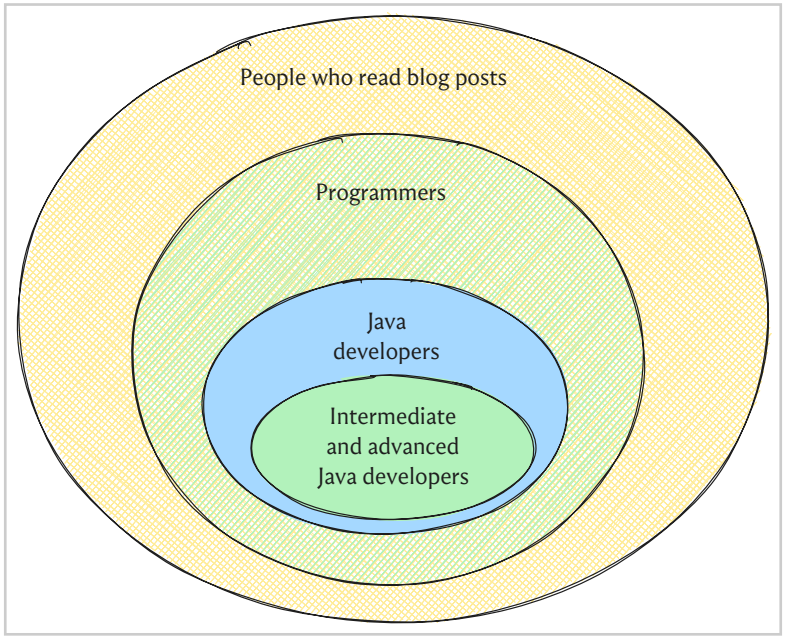
- Does this align with my goals?
- How much can I bring to the conversation?
- How many people are interested in this topic? And can I reach them?
- How long will this remain relevant?
- How difficult will this be to write?

2.3. Think one degree bigger

When you write an article, you hopefully have a type of reader in mind. For example, if you wrote an article called "Debugging Memory Leaks in Java," you probably assumed that the reader is an intermediate to advanced Java developer.

Most software bloggers never think to ask, "Is there a wider audience for this topic?"

For example, "intermediate to advanced Java developers" are a subset of "Java developers," who are a subset of "programmers," who are a subset of "people who read blog posts."



If you wrote an article for intermediate and advanced Java developers, how much would have to change for the article to appeal to Java developers of any experience level?

Often, the change is just an extra sentence or two early in the article to introduce a concept or replace jargon with more accessible terms.

> **Jeff**: Sony has a futuristic sci-fi movie they're looking to make.

> **Nick**: Cigarettes in space?
>
> **Jeff**: It's the final frontier, Nick.
>
> **Nick**: But wouldn't they blow up in an all-oxygen environment?
>
> **Jeff**: Probably. But it's an easy fix. One line of dialogue. "Thank God we invented the… you know, whatever device."
>
> — Thank You for Smoking (2005)

The set of all Java developers is about 10x larger than the set of intermediate and advanced Java developers. That means small tweaks can expand the reach of your article by an order of magnitude.

Obviously, you can't broaden every article, and you can't keep broadening your audience forever. No matter how well you explain background concepts, your tax accountant will never read an article about memory leaks in Java. The point isn't to write articles that appeal to every possible reader but to notice opportunities to reach a larger audience.

2.3.1. Example: "How I Stole Your Siacoin"

One of my earliest successes in blogging was an article called "How I Stole Your Siacoin". It was about a time I stole a reddit user's cryptocurrency (for noble reasons, I promise).

Initially, I thought the story would resonate with the few hundred people who followed a niche cryptocurrency called Siacoin. As I was editing the article, I realized that you didn't have to know anything about Siacoin to understand my story. I revised it slightly so it would make sense to cryptocurrency enthusiasts who had never heard of Siacoin.

Then, I realized I could even explain this story to people who knew nothing about cryptocurrency. I adjusted the terminology to use regular-person terms like "wallet" and "passphrase" and avoided crypto-specific terms like "blockchain" or "Merkle tree."

The article was my first ever hit. It became the most popular story of all time not only on the /r/siacoin subreddit but also on the larger /r/cryptocurrency subreddit. It reached the front page of Hacker News, even though readers there are generally hostile to cryptocurrency-focused stories.

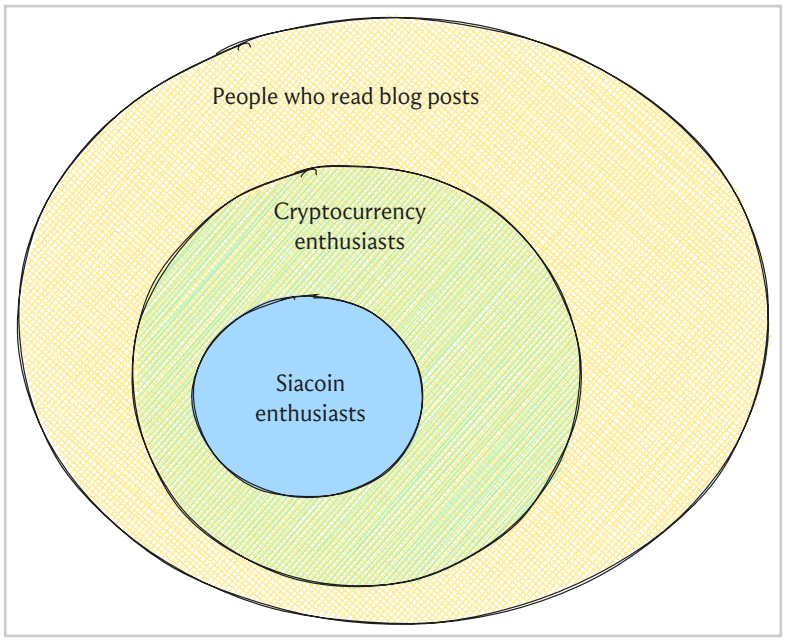


Figure 1. "How I Stole Your Siacoin" only needed a few tweaks to be enjoyable for people who didn't know anything about cryptocurrency.

2.4. Tell it like a story

A year after I started my blog, I hired a professional editor for feedback about my writing (I'll share more details in Work with a Professional Editor). She read my post about finding an inexpensive private chef through a gig worker platform and wrote back:

> What are you trying to accomplish with this story?
>
> If it's just the facts, you've done that, but I think you can do more. This is creative writing. Have fun with it. Make the reader laugh. Make the reader want to keep reading.

At first, I was confused. There was no "story." It was just a detailed explanation of a weird thing I did.

Oh, wait. That's a story!

It never occurred to me that I could use my blog to tell stories. I thought of blogging purely as a vehicle for explaining technical concepts. I tried to exclude my thoughts and feelings as much as possible.

But humans are wired for stories. Readers find stories more engaging, and they remember information better if it's part of a story.

In the nine years I've been blogging, a few of my posts have gone viral and attracted 100-

200k readers in a week. Uncoincidentally, those posts were all stories. When I meet someone at a tech conference that recognizes my name, it invariably turns out that they've read one of my stories.

2.4.1. What's a story?

My editor pointed out that my private chef post was just a collection of facts, not a story. So, what's the difference?

There's obviously more to storytelling than I can cover here, but I'll explain the basics of how I approach stories on my blog so they don't sound like bland dumps of information.

Stories have emotion

The most conspicuous difference between a story and a list of facts is that stories have emotion. Stories explain how you felt at different parts of an experience. Emotions are what allow the reader to relate to you and connect your experience to their life.

Consider this story that has zero emotion:

BAD: *Tell a story devoid of emotion.*

> I arrived at the interview at 9 AM. I shook the interviewer's hand. Then, I explained my credentials. My performance on the first question was weak. The second question was about my experience with PHP. For approximately 20 minutes, we discussed a hobby PHP interpreter I had written. My performance during the PHP portion of the interview was good.

Not a particularly engaging story, is it? It sounds like a robot who's impersonating a human (poorly).

What does the same story sound like with emotion?

GOOD: *Weave emotions into a story.*

> When I arrived at the interview, my palms were so sweaty that I probably drenched my interviewer's hand. I can't even recall his first question, just the panic I felt staring into his bored, expressionless face.
>
> At some point, I mumbled that I'd recently built a PHP interpreter from scratch, and his eyes lit up. Within seconds, I felt my anxieties melt away and my mind sharpen. Before I knew it, we'd been eagerly discussing the internals of my hobby PHP interpreter for 20 minutes.

The second story is more evocative. As you read it, you probably remembered a time in your life when you felt anxious or pressured to make a good impression. I bet the second version stays in your mind longer than the first one.

Stories zoom in on key moments

Instead of describing people or events in generalities, like "Nina is a cheerful teammate" or "Initech was a miserable place to work," look for opportunities to zoom in on a key moment that captures what you're describing.

BAD: *Tell a story using generalities.*

> My engineering director is bad with technology. He found our products confusing and would often need help achieving basic functionality with any piece of tech.

Instead, consider this version that zooms in on a particular moment:

GOOD: *Tell a story by zooming in on an important moment.*

> One morning, my engineering director approached my desk in a huff. He had discovered a defective button while testing his phone prototype, and he found this unacceptable. "Every time I push this button," he barked, "the thing immediately dies!" Anxiously, I inspected his device: he was pressing the power button.

The zoomed in version is more compelling and memorable. The reader can visualize this scene in their mind. If you simply list off facts about the engineering director, they just have to memorize details, which is harder and less fun.

Stories exclude irrelevant details

My friend Jessie is terrible at telling stories because she's fascinated by details that other people find irrelevant.

Jessie once told me a story about an older man who awkwardly tried to flirt with her at the grocery store. In the middle of this story, she explained at length how she got a great deal on peanut butter with her customer loyalty card.

As we approached minute two of this peanut butter digression, I lost patience. "Jessie, what happened with the creepy guy?" I asked. "The peanut butter isn't the interesting part."

"I saved $2." Jessie countered. "I think that's pretty interesting!"

Jessie's storytelling lacks editing. She shares every detail she can remember about her experience, regardless of how interesting it is for the listener.

When telling a story, think about what you want the reader to learn and what they'll find interesting. Ruthlessly cut anything that fails to serve those goals.

Stories grant creative license

A blog post is not court testimony. You didn't swear to tell the truth, the whole truth, and nothing but the truth. As an author, you can adjust the details of a story to fit your narrative. This is called "creative license."

You shouldn't make things up for an engaging story, but you also shouldn't feel compelled to disclose every single fact as accurately as possible. For example, in the previous section, I told a story about my friend Jessie, but I took several creative liberties:

- Her name isn't really Jessie.
- We had a falling out 10 years ago, so we're no longer friends.
- The setting was actually a pharmacy not a grocery store.
- The conversation occurred 20 years ago. I used direct quotes even though I don't remember the conversation verbatim.
- I exaggerated the length of the peanut butter digression.

Do you feel scandalized and betrayed? Hopefully not. I massaged details to make a tidier story, but the changes don't undermine the point of the story.

Use creative license according to your own sense of integrity. I use creative license to simplify messy-but-irrelevant details of a story, but I won't fabricate the essence of a story to make a point.

The type of article you're writing and the medium in which you publish also impacts how much creative license you have. If you publish a Rust tutorial on your personal blog and include a quirky story about your boss, Doofy McSpreadsheet, accidentally wiping the production database 12 times in a single year, the reader doesn't expect that story to be 100% true. If you write a scathing exposé about a public figure for a widely-read political blog, the reader will have a much stricter expectation that every detail is true.

2.4.2. When should you tell it like a story?

After my editor gave me the idea to use storytelling, I went a bit story-crazy. I tried to make *every* blog post into a story.

I wrote "How to Do Code Reviews Like a Human", which was a list of communication techniques for code reviews. But I finished the post and thought, "Oh, no! Nobody's going to like this post because it's not a story!" So, I awkwardly shoehorned in a story about my worst code review experience. I'm still proud of that post, but every time I read it, the story portion feels forced and out of place.

On the other hand, "TinyPilot: Build a KVM over IP for Under $100" is both a personal story and a tutorial, and the combination feels organic to me.

So, it's hard to set a simple rule for when to present information as a story other than, "Tell a story when it feels right." My mistake in "How to Do Code Reviews Like a Human" was including the story for the sake of including a story. In other posts where I mix a story with more practical information, the story serves the lesson, so it feels natural rather than forced.

2.5. Plan a path to your readers

Suppose you wrote the greatest beginner's tutorial imaginable for the Python programming language. Both your five-year-old nephew and 80-year-old dentist blazed through it with ease and delight. Everyone who reads your tutorial goes on to become a Python core contributor.

Bad news: nobody will ever read your Python tutorial.

"Lies!" you shout. "Thousands of developers learn Python every year. Why wouldn't my objectively awesome tutorial become popular?"

Well, think it through. What happens after you hit publish? How does anyone find your article?

You're probably thinking: Google.

Yes, your friend Google will index your tutorial and use its secret Google magic to identify your article's superior quality. Before you know it, your tutorial will be the top result for `python tutorial`.

Except that can't happen because there are so many Python tutorials out there already on sites that Google prefers over yours. You'll never even make it to the first page of results.

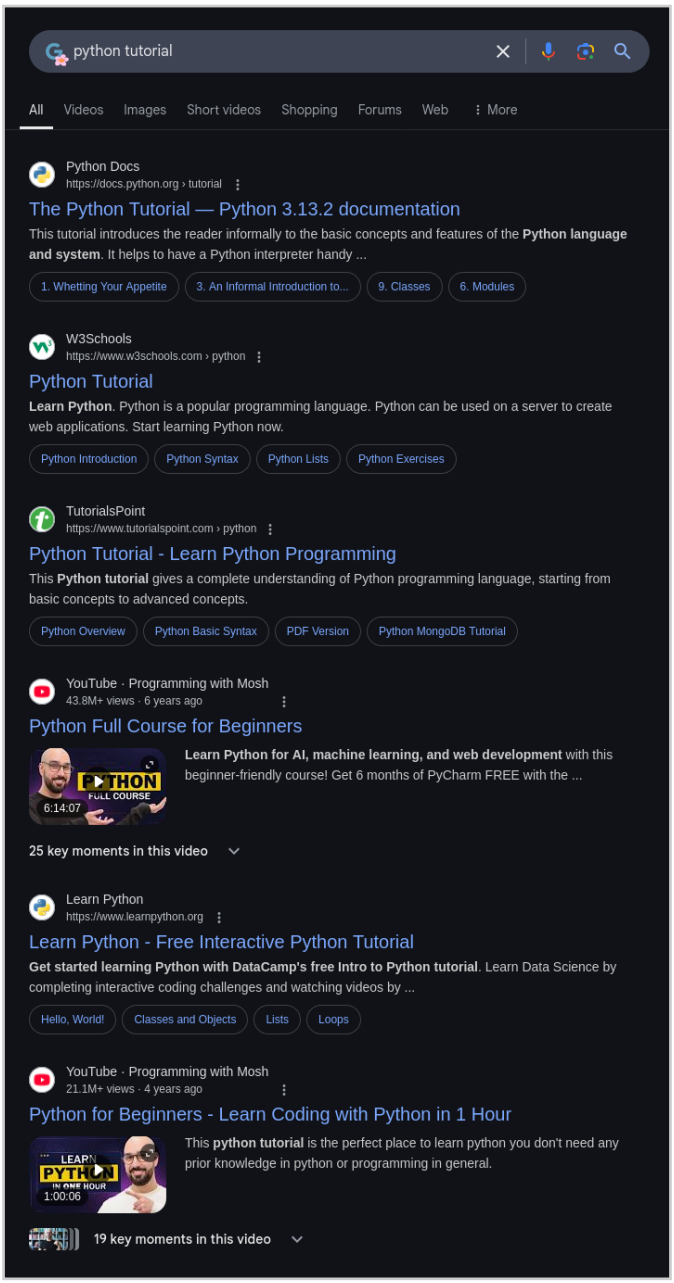


Figure 2. It's nearly impossible for a new blog post to rank well in Google for the search term `python tutorial`.

Okay, so you'll submit your Python tutorial to reddit. The /r/python subreddit has over 1.3 million subscribers. If even 5% of them read your article, that's a huge audience:

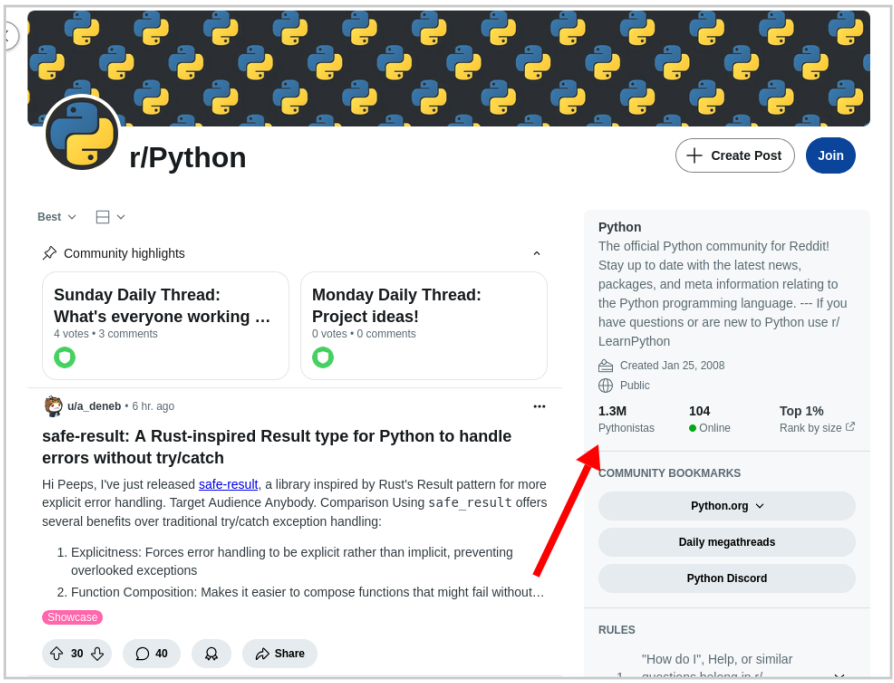


Figure 3. The /r/python subreddit has over 1.3 million subscribers.

Whoops! /r/python only accepts text posts, not external links, so you can't post your tutorial there.

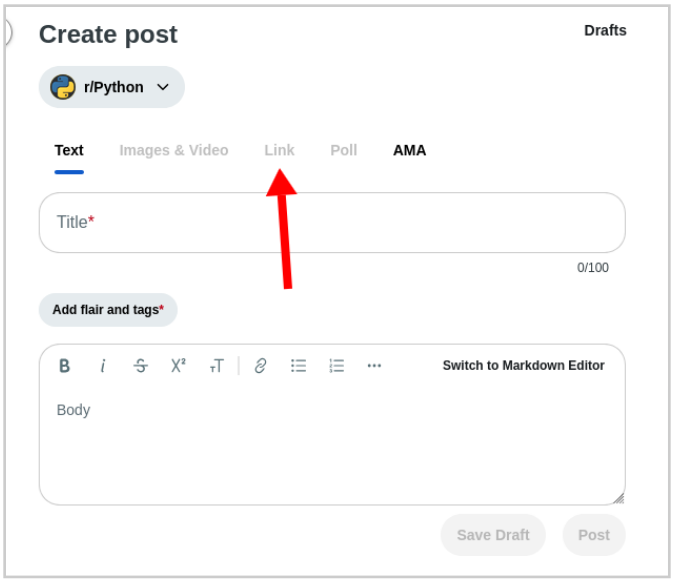


Figure 4. The /r/python subreddit disables the option to submit external links.

Fine, then you'll submit it to Hacker News. They accept anything and let their members decide what's interesting. Surely, they'll recognize the quality of your work!

Nope, it will flop there, too. Hacker News doesn't like tutorials, especially for mainstream technologies like Python.

You can try sharing your tutorial by tweeting it, skeeting it, or tooting it, but unless you already have a massive following on social media, that won't reach a critical mass either.

So, what's the answer? How do you get people to read your amazing Python tutorial?

The answer is that you don't write a beginner's Python tutorial.

2.5.1. You need a realistic path to your readers

If you want people to read your blog, choose topics that have a clear path to your readers. Before you begin writing, think through how readers will find your post.

Questions to ask when considering an article topic

- Is it realistic for readers to find you via Google search?
  - Are there already 500 articles about the same topic from more established websites?
  - What keywords would your target reader search? Try searching those keywords, and see whether there are already relevant results from well-known domains.
- If you're going to submit it to a link aggregator like Hacker News or Lobsters, how often do posts like yours succeed there?
- If you're going to share it on a subreddit or niche forum, does it have any chance there?
  - Does the forum accept links to blog posts?
    - The bigger the community, the stricter the rules tend to be about external links and self-promotion.
  - Do blog posts like yours ever succeed there?
  - Is the community still active?

The best plan is to give your post multiple chances to succeed. If you're betting everything on Google bubbling your post to the top, it could take months or years for you to find out if you succeeded. If you're relying on Hacker News or reddit to tell you whether your article is worth reading, they're going to break your heart a lot.

2.5.2. Example: "Using Zig to Unit Test a C Application"

In 2023, I wrote an article called "Using Zig to Unit Test a C Application.". It was about using a new low-level language called Zig to write tests for legacy C code.

Before I wrote the article, I knew that there were several places where I could share it. By luck, they all worked out:

- Hacker News is extremely friendly to Zig content, so my article reached the #7 spot on

the front page.

- Lobsters is extremely friendly to Zig content, so my article was one of the top links of the day.
- Google bubbled my article to the top result for the keywords `zig unit testing c`.
  - It's actually even a top result for just `zig unit testing` because there aren't many articles about the topic.
- The /r/Zig subreddit accepts links to blog posts, even if they're self-promotion, so my post reached the top spot in that subreddit.
- Ziggit is a niche forum that's welcoming to Zig-related articles, so my post received 1,000 views from Ziggit.

2.6. Show more pictures

The biggest bang-for-your-buck change you can make to a blog post is adding pictures.

If your article features long stretches of text, think about whether there's any photo, screenshot, graph, or diagram that could make the post more visually interesting.

- If you're talking about a program with a graphical interface, show screenshots.
- If you're talking about an improvement in metrics like app performance or active users, show graphs.
- If you're writing about your server getting overloaded, show a screenshot of what that looked like in your dashboard or email alerts.
- If you're explaining a difficult concept, draw a diagram.

I hire illustrators for most of my blog posts. I typically pay $50-100 per illustration. For simple diagrams like the nested circle sketches above, I use Excalidraw, which is free and open-source.

Free stock photos and AI-generated images are better than nothing, but they're worse than anything else, including terrible MS Paint drawings.

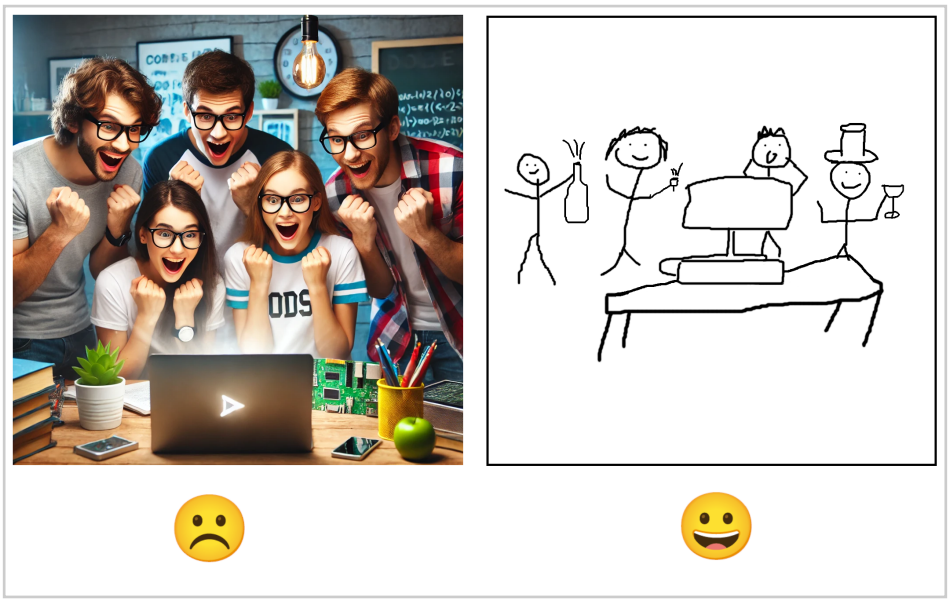


Figure 5. Even a terrible MS Paint drawing is more interesting than an AI-generated image.

2.7. Accomodate skimmers

Many readers skim an article first to decide if it's worth reading. Dazzle those readers during the skim.

If the reader only saw your headings and images, would it pique their interest?

The worst thing for a skimmer to see is a wall of text: long paragraphs with no images or headings to break them up. Just text, text, text all the way down.

2.7.1. Example: Boring structure vs. interesting structure

I wrote "End-to-End Testing Web Apps: The Painless Way" in 2019, before I thought about structure. If you skim the article, does it make you want to read the full version?

Probably not. The headings don't reveal much about the content, and the visuals are confusing.

Consider my more recent article, "I Regret My $46k Website Redesign."

If you skim that article, you still see the bones of a good story, and there are interesting visual elements to draw the reader in.

One of those articles barely attracted any readers, and the other became one of the most popular articles I ever published, attracting 150k unique readers in its first week. Can you

guess which is which?

2.7.2. Tool: Read like a skimmer

The *Refactoring English* website contains a JavaScript bookmarklet that you can use to see what your article looks like with just headings and images.

Chapter 3. Good vs. Bad Content Marketing

Blogging is a great way to find customers for your product. If you do it well, a single blog post can bring you hundreds of initial customers and attract attention for years. Best of all, you can do it without spending any money. I've found early customers for all of my products through blogging.

The popular term for advertising via blogging is "content marketing." I avoid that term because it makes the writing sound bland and soulless, as if it exists to trick the users into consuming an ad.

Most of the people who talk about content marketing don't care about the content part of the equation at all. They'd publish dancing poop emoji if it drove signups to their product.

Blogging to attract customers doesn't have to be soulless or self-serving. When you write something that's actually worth the reader's time, you not only attract attention to your product, but you earn credibility with the reader.

Imagine that you were in charge of picking a tool to support automated testing for your team. One product has a dazzling website, and the other has a more austere site but an excellent blog about effective testing techniques. I'd absolutely choose the company with the blog, as it indicates that the team understands the domain and designed their product with sound principles in mind.

3.1. Bad content marketing

The bad way to do content marketing is to just write about how great your product is.

As an example, imagine that you created a to-do list app for developers, and you call it DevDo. Here are some bad article titles for attracting customers:

- "The Five Coolest Features of DevDo"
- "Feeling Overwhelmed by Your To Do List? DevDo is the Only Solution"
- "Here Are 2000 words of AI-Generated Blather About DevDo"

Those articles have a low chance of success because readers would (correctly) perceive them

as ads. If you submitted them to blog discovery sites like Hacker News, Lobsters, or /r/programming, users would likely downvote or flag your post as spam.

Those articles bring no value to the reader. They exist solely to talk about how great your product is. You might argue that boasting about your product *does* help the reader—they'll discover a useful app! But a piece of media that exists solely to raise awareness of a product is called an ad.

3.2. Good content marketing

So, if product brags are bad content marketing, what's the alternative?

Good content marketing has two important qualities:

1. It provides value to the reader even if they never use your product.
2. It highlights your product organically rather than mentioning it for the sake of advertising.

Instead of focusing on your product and looking for ways to write about it, start with your target customers and think about what they'd like to learn.

Consider how your reader will find your article. They're either going to come across the post on a link sharing site like Hacker News or reddit, or they're going to search for a topic. But your target reader doesn't know about your product yet, so they're not going to search for your product by name (see Plan a path to your readers).

For DevDo, your target customers are software developers who want a better solution for managing their tasks. That means you should brainstorm ideas around the topics of software development or task management.

3.2.1. Brainstorming developer-oriented topics

To find topics that appeal to software developers, think about what you learned as you built your product:

- Did you discover a new technique or tool?
- Did you discover surprising behavior in technology that you regularly use?
- Did you find an unsusual use case for an existing technology?
    - e.g., you used SQLite as a message broker
- What do you wish someone had explained to you better when you started developing your app?

The more surprising, the better, but you don't have to find anything groundbreaking. You can write, "Reflections on Using SQLite in Production for the First Time," and that will do well as long as you share honest, useful details of your experience.

Your blog post shouldn't pitch your product to the reader. Instead, it should sound like how you'd describe your work in a job interview. Describe your app's goals, constraints, and the thought process for your engineering decisions.

BAD: *Pitch your product to the reader.*

> Lightning-fast responsiveness is what makes **DevDo** users feel 100% satisfied with my product, so I always used `int8` in places where I didn't need a full 32-bit integer. Bloated alternatives like Asana and Microsoft To-Do are so sloppy and bloated that they probably default to `int64`. Click **here** to save 10% on your subscription to **DevDo**!

GOOD: *Describe the engineering challenges of building your product.*

> One of my key requirements was to display DevDo's main task list to the user in under 200ms. I created a continuous integration job that ran on every commit and recorded the page load time. I was surprised to find that changing `int` (implicitly 32 bits) to `int8` (8 bits) on one of my core data structures sped up the initial page load by 18%.

3.2.2. Brainstorming task management topics

The other blogging angle for a to-do list app is to write about task management.

If you care enough about task management to write your own to-do list app, you probably have a unique perspective, so write about that:

- What do you passionately believe about task management that nobody else articulates well?
- What's a task management technique that other apps support poorly?
- How did DevDo change your approach to task management?

As with topics to appeal to developers, the article shouldn't pitch DevDo to the reader. It should share ideas that are useful even if the reader never uses your product. You're talking about a technique or idea where DevDo is an implementation detail, not the solution itself.

BAD: *Pitch your product to the reader.*

> One of the things I always hated about other to-do list apps was managing projects with multiple substeps. For example, if I book a hotel, I first have to research options, then review them with my spouse, then make a reservation.
>
> Some to-do apps would overwhelm me with all the steps at once. Others show one task at a time, but as soon as I check off one task, the next task would appear in its place,

> making me feel like I hadn't made any progress.
>
> I designed DevDo to serve this use case better, so it lets you configure how many subtasks of the project to show per day and how long to wait before showing the next subtask.

GOOD: *Focus on the technique rather than your product.*

> I always break large or complex tasks into a series of substeps that occur in order. For example, if I book a hotel, I first have to research options, then review them with my spouse, then make a reservation. I get overwhelmed if I see all the tasks at once, and I find it demoralizing if checking off an item immediately adds a new item to my list.
>
> So, for multi-task projects, I configure DevDo to show me only one task at at a time and to wait a day before bumping the next task onto my list of outstanding tasks. <show screenshot of this setting in DevDo> I've done this in other apps by splitting the tasks by date, so that task A is due Monday, task B is due Tuesday, etc.

In the bad version, the focus is on the app rather than the technique of limiting projects to one task at a time. In the good version, the focus is on the technique. It shows how to apply this with DevDo, but it also explains how you can replicate it with other apps.

The tone isn't, "Look at this cool feature of my app!" It's matter of fact - you use this feature to solve your problem. The focus is the technique, not your product.

3.3. Tactfully include your product

It takes a bit of finesse to find the right amount of product mentions in a blog post. You don't want to mention your product so frequently that your article feels like an ad, but you also don't want to put all that effort into writing only for readers to finish your article and not even realize you offer a related product.

A good blog post about your product should make the reader curious rather than manipulated or tricked. Make sure it's easy for the reader to explore that curiosity and learn more about your product. Link to your product page the first time you mention it, and link again at the end of your post.

3.4. Case study: Finding customers for TinyPilot through blogging

My biggest success in finding customers through blogging was with my 2020 post, "TinyPilot: Build a KVM over IP for Under $100." I'd created a hardware device that let users control their computers remotely, but I wasn't sure anyone would want to buy it. Within four hours of publishing the blog post, customers purchased my entire stock. The article reached 22k readers on its first day, and 59k in its first week. Even five years later, about 500 readers per month read that post.

The post told the story of how I built the prototype, and I taught the reader to build an identical device with off-the-shelf parts. At the end of the post, I included a link to buy a pre-packaged kit from me rather than finding the parts from scattered merchants.

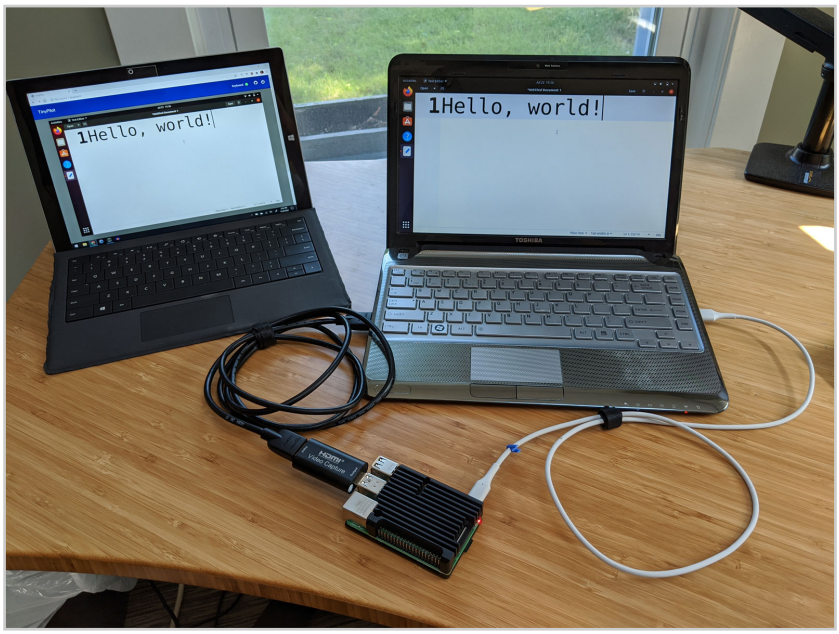


Figure 6. Photo from my blog post about creating TinyPilot, a device that allowed customers to control computers remotely.

My TinyPilot post worked because it delivered value to the reader before trying to sell them something. My paid product offered something extra.

After the initial launch post, I continued finding new TinyPilot customers through blogging. I created TinyPilot to help me manage my home servers, so I looked for other ways to write about home servers. Later, I wrote "Building a Budget Homelab NAS Server," which explained how I chose parts and built my home storage server. When I cover installing the operating system, I show screenshots of what that step looks like in TinyPilot:

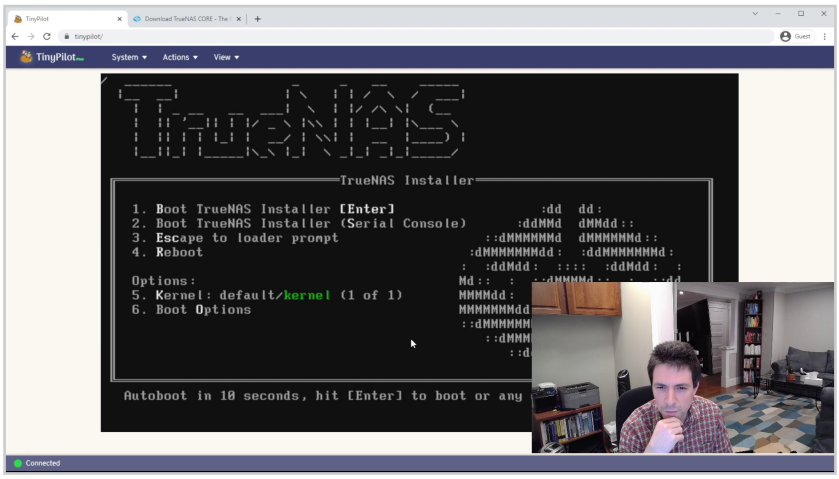


Figure 7. Screenshot featuring my TinyPilot product in a post about creating a home storage server.

For people who manage home servers, they know the operating system install step is a pain. If you don't have an extra keyboard and monitor lying around, you need to borrow the setup from your main workstation, which means crawling under your computer, swapping around a bunch of cables, then doing everything in reverse after the install is complete.

I didn't pitch TinyPilot to the reader, but I wanted them to see the screenshots and think, "Michael can install an OS remotely through his browser? That's neat. What's this TinyPilot thing he's using to do that?"

3.5. Case study: Tailscale explains NAT traversal

Tailscale is networking software that makes it easy to create secure, private connections between computers over the Internet. They enjoy a positive reputation among developers, in no small part due to their high-quality technical blog posts. One example is "How NAT traversal works," which explains the complexity you encounter if you try to create a direct network connection between two computers over the Internet.

The bad content marketing version of this article would be called "Never Attempt Peer-to-Peer Connections without Tailscale," and they'd go over all the reasons why NAT traversal is hard, so you should use Tailscale instead.

Instead, Tailscale wrote an extremely thorough and accessible explanation of network address translation (NAT). The article frequently mentions Tailscale but in a way that's organic and helpful rather than forced and spammy. They explain how the Tailscale team thought about these problems when creating the product, which establishes Tailscale's team in the reader's mind as knowledgeable and thoughtful.

If you Google "NAT traversal," Tailscale's blog post is one of the top results, and none of the others even come close in terms of quality or thoroughness. If you're someone who needs to understand NAT traversal, but you've never heard of Tailscale, the blog post will thoroughly

explain Tailscale, and you'd also be curious about this company that simplifies the problem.

3.6. Case study: Basecamp captures hearts and minds of small agencies

37signals is a software company that built their brand through skillful content marketing. For eight years, they ran a blog called *Signal v. Noise* that enjoyed wide popularity among tech readers.

The 37signals blog covered a variety of topics, but their most popular posts were the ones that aggressively criticized big tech practices and attitudes:

- "We only hire the best"
  - Critiques the arrogance of big tech companies who mistakenly believe they have the best engineers.
- "Eat, sleep, code, repeat" is such bullshit
  - Attacks the "Eat, sleep, code, repeat" tagline at a Google developer conference as hostile to work-life balance.
- The open-plan office is a terrible, horrible, no good, very bad idea
  - You can probably guess.

So, what's the point of attacking big tech?

37signals's most popular commercial product is project management software called Basecamp. They explicitly market Basecamp to "smaller, hungrier businesses, not big, sluggish ones."

By writing provocative blog posts, 37signals generated attention on sites like Hacker News, Twitter, and reddit. The posts earned credibility and respect from the small companies in their target market.

Chapter 4. Rules for Writing Software Tutorials

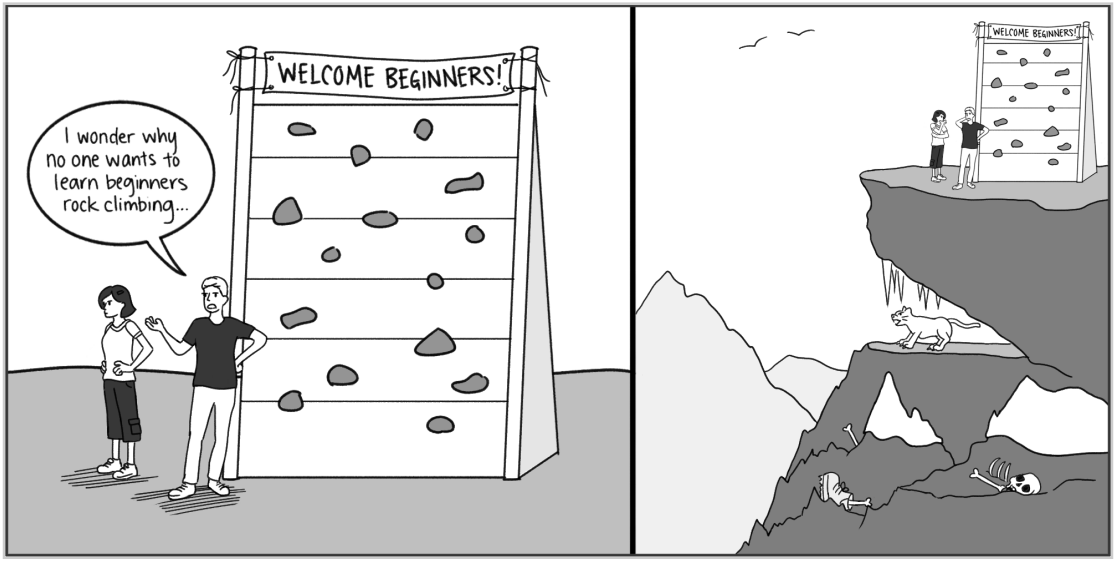
Most software tutorials are tragically flawed.

Tutorials often forget to mention some key detail, preventing readers from replicating the author's process. Other times, the author brings in hidden assumptions that don't match their readers' expectations.

The good news is that it's easier than you think to write an exceptional software tutorial. You can stand out in a sea of mediocre guides by following a few simple rules.

4.1. Write for beginners

The most common mistake tutorials make is explaining beginner-level concepts using expert-level terminology.



Most people who seek out tutorials are beginners. They may not be beginners to programming, but they're beginners to the domain they're trying to learn about.

BAD: *Refer to concepts that beginners won't understand.*

> In this tutorial, I'll show you how to create your first "Hello world" SPA using React.

> Open the included `hello.jsx` file and change the greeting from `"Hello world"` to `"Hello universe"`.
>
> The browser should hot reload with the new text. Because of React's efficient JSX transpilation, the change feels instant.
>
> The browser doesn't even have to soft reload the page because React's reconciliation engine compares the virtual DOM to the rendered DOM and updates only the DOM elements that require changes.

The above example would confuse and alienate beginners.

A developer who's new to the React web framework won't understand terms like "JSX transpilation" or "reconciliation engine." They probably also won't understand "SPA," "soft reload," or "virtual DOM" unless they've worked with other JavaScript frameworks.

When you're writing a tutorial, remember that you're explaining things to a non-expert. Avoid jargon, abbreviations, or terms that would be meaningless to a newcomer.

Here's an introduction to a React tutorial that uses language most readers will understand, even if they have no background in programming:

GOOD: *Use terms that make sense to beginners.*

> In this tutorial, I'll show you how to create a simple webpage using modern web development tools.
>
> To generate the website, I'm using React, a free and popular tool for building websites.
>
> React is a great tool for creating your first website, but it's also full-featured and powerful enough to build sophisticated apps that serve millions of users.

Writing for beginners doesn't mean alienating everyone with more experience. A knowledgeable reader can scan your tutorial and skip the information they already know, but a beginner can't read a guide for experts.

4.2. Promise a clear outcome in the title

If a prospective reader is Googling a problem, would the title of your article lead them to the solution? If they see your tutorial on social media or in a newsletter, will your title convince them it's worth clicking?

Consider the following weak titles:

BAD: *Use vague titles.*

> - A Complete Guide to Becoming a Python CSV Ninja
> - How to Build Your Own Twitter
> - Key Mime Pi: A Cool Gadget You Can Make
> - How to Make a Compiler

The above examples are poor titles because they're vague. From the titles alone, you'd be hard-pressed to say what they'd teach you.

A tutorial's title should explain succinctly what the reader can expect to achieve by following your guide.

Here are clearer rewrites of the previous titles:

GOOD: *Use titles that promise a clear outcome.*

> - How to Read a CSV File in Python
> - Build a Real-Time Twitter Clone in 15 Minutes with Phoenix LiveView
> - Key Mime Pi: Turn Your Raspberry Pi into a Remote Keyboard
> - How to Write a C Compiler in 500 Lines of Python

These titles give you a clear sense of what you'd learn by reading the tutorial. The titles are clear and specific in what the tutorial delivers.

4.3. Explain the goal in the introduction

If the reader clicks your tutorial, you're off to a great start. Someone is interested in what you have to say. But you still have to convince them to continue reading.

As the reader begins a tutorial, they're trying to answer two critical questions as quickly as possible:

1. Should I care about this technology?
2. If I care, is this the right tutorial for me?

The first few sentences of your article should answer those questions.

For example, if you were writing a tutorial about how to use Docker containers, this would be a terrible introduction:

> **An Introduction to Docker Containers**
>
> Docker is an extremely powerful and versatile technology. It allows you to run your app in a container, which means that it's separate from everything else on the system.
>
> In this tutorial, I'll show you how to use Docker to run containers on your internal infrastructure as well as in the cloud.

Based on the above introduction, what problem does Docker solve? Who should use it?

The introduction fails to answer those questions and instead hand-waves with vague terms that ignore anything the reader cares about.

Here's a rewrite that explains how Docker solves pain points the reader might have:

GOOD: *Explain a concrete outcome and benefit.*

> **How to Use Docker for Reliable App Deployments**
>
> Do you have a production server that you're terrified to touch because nobody knows how to rebuild it if it goes offline? Have you ever torn your hair out trying to figure out why your staging environment behaves differently than your production environment?
>
> Docker is a tool for packaging your app so that it has a consistent, reproducible environment wherever it runs. Docker allows you to define your app's environment and dependencies in source code, so you know exactly what's there, even if your app has survived years of tweaks by different teams.
>
> In this tutorial, I'll show you how to use Docker to package a simple web app and help you avoid common Docker gotchas.

The above introduction explains the problems Docker solves and what the tutorial will deliver.

The introduction doesn't say, "This tutorial is for people who are brand new to Docker," but it doesn't need to. It introduces Docker as a new concept, which tells the reader that the guide is for newcomers.

4.4. Show the end result

As soon as possible, show a working demo or screenshot of what the reader will create by the end of your tutorial.

The end result doesn't have to be anything visually stunning. Here's an example of how I

showed the terminal UI the user would see at the end of my tutorial:

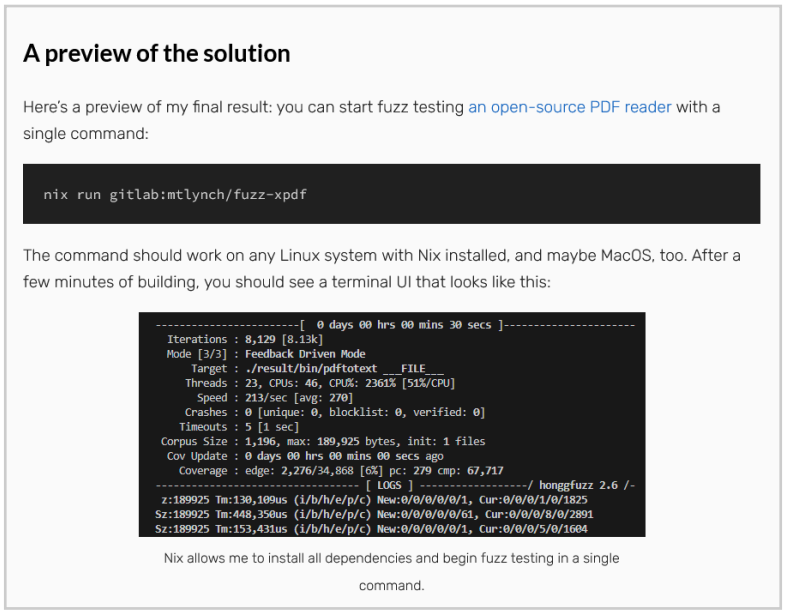


Figure 8. Early in your tutorial, show the reader a preview of what they'll produce by the end.

Showing the final product reduces ambiguity about your goal. It helps the reader understand if it's the right guide for them.

4.5. Make code snippets copy/pasteable

As the reader follows your tutorial, they'll want to copy/paste your code snippets into their editor or terminal.

An astonishing number of tutorials unwittingly break copy/paste functionality, making it difficult for the reader to follow along with their examples.

4.5.1. Make shell commands copyable

One of the most common mistakes authors make in code snippets is including the shell prompt character.

A shell snippet with leading `$` characters will break when the user tries to paste it into their terminal.

```
$ sudo apt update       # <<< Don't do this!
$ sudo apt install vim  # <<< Users can't copy/paste this sequence without
$ vim hello.txt         # << picking up the $ character and breaking the command.
```

Even Google gets this wrong. In some places, their documentation helpfully offers a "Copy code sample" button.

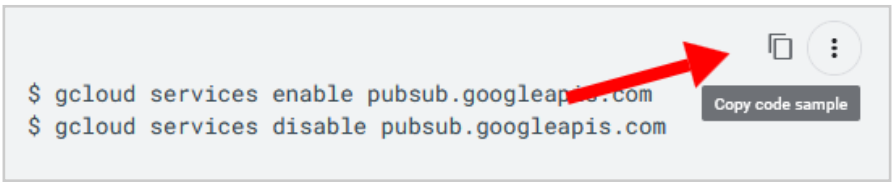


Figure 9. Google offers a "Copy code sample" button that incorrectly copies shell terminal characters.

If you click the copy button, it copies the `$` terminal prompt character, so you can't paste the code:

```
michael@ubuntu: $ gcloud services enable pubsub.googleapis.com
$ gcloud services disable pubsub.googleapis.com
bash: $: command not found
bash: $: command not found
michael@ubuntu:
```

There's a different version of this copy/paste error that's more subtle:

BAD: *Present a sequence of commands that require user input.*

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.9
```

If I try to paste the above snippet, here's what I see in the terminal:

```
0 upgraded, 89 newly installed, 0 to remove and 2 not upgraded.
Need to get 36.1 MB of archives.
After this operation, 150 MB of additional disk space will be used.
Do you want to continue? [Y/n] Abort.
$
```

What happened?

When the `apt install software-properties-common` command executes, it prompts the user for input. The user can't answer the prompt because `apt` just continues reading from the clipboard paste.

Most command-line tools offer flags or environment variables to avoid forcing the user to respond interactively. Use non-interactive flags to make command snippets easy for the user to paste into their terminal.

GOOD: *Use command-line flags that avoid interactive user input.*

```
sudo apt update
sudo apt install --yes software-properties-common
sudo add-apt-repository --yes ppa:deadsnakes/ppa
sudo apt install --yes python3.9
```

4.5.2. Join shell commands with `&&`

Take another look at the Python installation example I showed above, as it has a second problem:

BAD: *Ignore failing commands.*

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.9
```

If one of the commands fails, the user might not notice. For example, if the first command was **`sudo apt cache ppa:dummy:non-existent`**, that command would fail, but the shell would happily execute the next command as if everything was fine.

In most Linux shells, you can join commands with **`&&`** and continue lines with a backslash. That tells the shell to stop when any command fails.

Here's the user-friendly way to include a series of copy-pasteable commands:

GOOD: *Chain commands together with* **`&&`**.

```
sudo apt update && \
  sudo apt install --yes software-properties-common && \
  sudo add-apt-repository --yes ppa:deadsnakes/ppa && \
  sudo apt install --yes python3.9
```

The user can copy/paste the entire sequence without having to tinker with it in an intermediate step. If any of the commands fail, the sequence stops immediately.

4.5.3. Only show the shell prompt to demonstrate output

Occasionally, showing the shell prompt character benefits the reader.

If you show a command and its expected output, the shell prompt character helps the reader distinguish between what they type and what the command returns.

For example, a tutorial about the `jq` utility might present results like this:

GOOD: *Use the shell prompt character to distinguish between a command and its output.*

> The `jq` utility allows you to restructure JSON data elegantly:

```
$ curl \
  --silent \
  --show-error \
  https://status.supabase.com/api/v2/summary.json | \
  jq '.components[] | {name, status}'
{
  "name": "Analytics",
  "status": "operational"
}
{
  "name": "API Gateway",
  "status": "operational"
}
...
```

4.5.4. Exclude line numbers from copyable text

It's fine to include line numbers alongside your code snippets, but make sure they don't break copy/paste. For example, if the user tries to copy the `count_tables` function from the following snippet, they'd have to remove line numbers from their pasted text.

BAD: *Include line numbers in copyable text.*

```
123 def count_tables(form):
124   if not form:
125     return None
```

4.6. Use long versions of command-line flags

Command-line utilities often have two versions of the same flag: a short version and a long version.

```
-r / --recursive : Run recursively
^      ^
|      |
|      long flag
```

```
|
short flag
```

Always use long flags in tutorials. They're more descriptive, so they make your code easier to read, especially for beginners.

BAD: *Use short, opaque versions of command-line flags.*

> Run the following command to find all the pages with `<span>` elements:
>
> ```
> grep -i -o -m 2 -r '<span.*</span>' ./
> ```

Even if the reader is familiar with the `grep` tool, they probably haven't memorized all of its flags.

Use long flags to make your examples clear to both experienced and inexperienced readers.

GOOD: *Use verbose, descriptive versions of command-line flags.*

> Run the following command to find all the pages with `<span>` elements:
>
> ```
> grep \
>   --ignore-case \
>   --only-matching \
>   --max-count=2 \
>   --recursive \
>   '<span.*</span>' \
>   ./
> ```

4.7. Separate user-defined values from reusable logic

Often, a code example contains elements that are inherent to the solution and elements that each reader can customize for themselves. Make it clear to the reader which is which.

The distinction between a user-defined value and the rest of the code might seem obvious to you, but it's unclear to someone new to the technology.

4.7.1. Use environment variables in command-line examples

A logging service that I use lists the following example code for retrieving my logs:

BAD: *Bake user-defined variables into the code.*

```
LOGS_ROUTE="$(
  curl \
    --silent \
    --header "X-Example-Token: YOUR-API-TOKEN" \
    http://api.example.com/routes \
    | grep "^logs " \
    | awk '{print $2}'
    )" && \
  curl \
    --silent \
    --header "X-Example-Token: YOUR-API-TOKEN" \
    "http://api.example.com${LOGS_ROUTE}" \
    | awk \
        -F'T' \
        '$1 >= "YYYY-MM-DD" && $1 <= "YYYY-MM-DD" {print $0}'
```

Given that example, which values am I supposed to replace?

Clearly, **`YOUR-API-TOKEN`** is a placeholder that I need to replace, but what about **`YYYY-MM-DD`**? Am I supposed to replace it with real dates like **`2024-11-23`**? Or is it specifying a date schema, meaning that **`YYYY-MM-DD`** is the literal value I'm supposed to keep?

There are several other numbers and strings in the example. Do I need to replace any of those?

Instead of forcing the reader to search through your example and guess which values to change, create a clean separation. Start with the editable values, then give them the snippet they can copy/paste verbatim.

Here's my rewrite of the example above:

GOOD: *Use environment variables for user-defined values.*

```
API_TOKEN='pk-example-key' # Replace with your API key, which
                          # always has the prefix "pk-".
START_DATE='2024-01-01'   # Replace with desired start date.
END_DATE='2024-12-31'     # Replace with desired end date.

LOGS_ROUTE="$(
  curl \
    --silent \
    --header "X-Example-Token: $API_TOKEN" \
    http://api.example.com/routes \
    | grep "^logs " \
    | awk '{print $2}' \
    )" && \
  curl \
    --silent \
    --header "X-Example-Token: $API_TOKEN" \
```

```
    "http://api.example.com${LOGS_ROUTE}" \
    | awk \
      -F'T' \
      -v start="$START_DATE" \
      -v end="$END_DATE" \
      '$1 >= start && $1 <= end {print $0}'
```

The new version distinguishes between values the reader must replace and code that must remain in place.

Using environment variables clarifies the intent of the user-defined values and means the user only has to enter each unique value once.

4.7.2. Use named constants in source code

Suppose that you were writing a tutorial that demonstrated how to crop an image so that it displays well in social sharing cards on Bluesky, Twitter, and Facebook:

Figure 10. The image in the post above has image dimensions specifically to fit social sharing cards.

Here's how you might show code for cropping an image to fit social media cards:

BAD: *Make the reader guess which numbers are changeable.*

```
func CropForSocialSharing(img image.Image) image.Image {
  targetWidth := 800
  targetHeight := int(float64(targetWidth) / 1.91)
  bounds := img.Bounds()

  x := (bounds.Max.X - targetWidth) / 2
  y := (bounds.Max.Y - targetHeight) / 2

  rgba := image.NewRGBA(
    image.Rect(x, y, x+targetWidth, y+targetHeight))
  draw.Draw(
```

```
    rgba, rgba.Bounds(), img, image.Point{x, y}, draw.Src)
  return rgba
}
```

The example shows four numbers:

- `800`
- `1.91`
- `2`
- `2` (again)

Which numbers are the reader free to change?

In source code examples, make it obvious which values are inherently part of the solution and which are arbitrary.

Consider this rewrite that makes the intent of the numbers clearer:

GOOD: *Make unchangeable values explicit.*

```
// Use a 1.91:1 aspect ratio, which is the dominant ratio
// on popular social networking platforms.
const socialCardRatio = 1.91

func CropForSocialSharing(img image.Image) image.Image {
  // I prefer social cards with an 800px width, but you can
  // make this larger or smaller.
  targetWidth := 800

  // Choose a height that fits the target aspect ratio.
  targetHeight := int(float64(targetWidth) / socialCardRatio)

  bounds := img.Bounds()

  // Keep the center of the new image as close as possible to
  // the center of the original image.
  x := (bounds.Max.X - targetWidth) / 2
  y := (bounds.Max.Y - targetHeight) / 2

  rgba := image.NewRGBA(
    image.Rect(0, 0, targetWidth, targetHeight))
  draw.Draw(
    rgba, rgba.Bounds(), img, image.Point{x, y}, draw.Src)

  return rgba
}
```

In this example, the code puts the value of `1.91` in a named constant and has an accompanying comment explaining the number. That communicates to the reader that they

shouldn't change the value, as it will cause the function to create images with poor proportions for social sharing cards.

On the other hand, the value of `800` is more flexible, and the comment makes it obvious to the reader that they're free to choose a different number.

4.8. Use unambiguous example values

In code examples, use variable names and values that make it obvious to the reader that they're examples. Avoid using names or values that the reader might mistake for language keywords or library APIs.

For example, I've seen multiple database library READMEs show code that looks like this:

BAD: *Use example values that look like language keywords.*

> Create a SQLite database table with the following commands:

```
create table tbl(id, column);
insert into tbl(0,root);
```

The field names and values make this example extremely confusing to readers who are unfamiliar with SQLite's query syntax.

- Is the name of the table `table` or `tbl`?
- Is `id` a required field that every table must have?
- Is `column` a SQLite keyword? Or does the code literally create a column named `column`?

Consider this alternative example that chooses names and values that are unambiguous.

GOOD: *Use example values that are obviously examples.*

> Create a SQLite database table with the following commands:

```
-- Create a table in the database to store pets' names
-- and favorite foods.
CREATE TABLE pets (
  pet_name TEXT NOT NULL,
  favorite_food TEXT NOT NULL
);

-- Add a pet to the table named Skippy whose favorite
-- food is Bacon Treats.
```

```
INSERT INTO pets
VALUES ('Skippy', 'Bacon Treats');
```

No reader is going to think that `'Skippy'` or `'Bacon Treats'` are SQLite keywords. `pet_name` is obvious as something we're defining in our particular table and not a feature of SQLite.

The example also takes extra steps to delineate the boundaries between language keywords, schema definition, and example data:

- It uses different casing to distinguish language keywords (`INSERT INTO`) and user-defined values (`pets`).
- It adds comments to further clarify which values are user-defined.
- It uses verbose language features to make the role of the user-defined values more obvious.

4.8.1. Use example values that looks like real-world data

A common anti-pattern in tutorials is choosing uncreative values for variables that effectively just describe the data type.

BAD: *Use lazy example values that echo the data type.*

```
message = string('string')
filePath = FilePath('folder/file')
username = User('user')
```

A string value of `'string'` creates ambiguity for the reader because they'll wonder whether `'string'` is meaningful within the language or if the author was simply lazy in picking a value.

Instead, choose example values that stand out conspicuously as example data:

GOOD: *Use examples values that look like real-world values.*

```
message = string('Hello, world!')
filePath = FilePath('photos/Italy/me-high-fiving-pope.jpg')
username = User('mike1234')
```

Software blogger Thorsten Ball calls this technique "[using data that looks like data."]

It's fine to theme your examples to a TV show or movie, but consider whether it will confuse readers unfamiliar with what you're referencing. Any reader can recognize `"Jim Halpert"` as a person's name even if they haven't seen *The Office*. But if you're a *Star Trek* fan, please

don't use `"Data"` as an example name, as it's quite confusing to people who don't know the show.

4.9. Spare the reader from mindless tasks

The reader will appreciate your tutorial if you show that you respect their time.

Don't force the reader to perform tedious interactive steps when a command-line snippet would achieve the same thing.

BAD: *Force the reader to perform unnecessarily tedious steps.*

> Do the following tedious steps:
>
> 1. Run `sudo nano /etc/hostname`
> 2. Erase the hostname
> 3. Type in `awesomecopter`
> 4. Hit Ctrl+o to save the contents
> 5. Hit Ctrl+x to exit the editor

The above steps make your tutorial boring and error-prone. Who wants to waste mental cycles manually editing a text file?

Instead, show the reader a command-line snippet that achieves what they need:

GOOD: *Script the steps that are not informative or interesting.*

> Paste the following simple command:
>
> ```
> echo 'awesomecopter' | sudo tee /etc/hostname
> ```

4.10. Keep your code in a working state

Some authors design their tutorials the way you'd give instructions for an origami structure. It's a mysterious sequence of twists and folds until you get to the end, and then: wow, it's a beautiful swan!

A grand finale might be fun for origami, but it's stressful for the reader.

Give the reader confidence that they're following along correctly by keeping your example code in a working state.

BAD: *Reference code that the reader hasn't yet seen.*

> Here's some example code, but don't even think about compiling it. It's missing the `parseOption` function LOL!

```
// example.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 256

int main() {
    char line[MAX_LINE_LENGTH];
    char key[MAX_LINE_LENGTH];
    char value[MAX_LINE_LENGTH];

    while (fgets(line, sizeof(line), stdin)) {
      // Don't do this!
      parseOption(line, key, value); // <<< Not yet defined

      printf("Key: '%s', Value: '%s'\n", key, value);
    }

    return 0;
}
```

If the reader tries to compile the above example, they get an error:

```
$ gcc example.c -o example
example.c: In function 'main':
example.c:14:7: warning: implicit declaration of function 'parseOption'
  [-Wimplicit-function-declaration]
   14 |       parseOption(line, key, value); // <<< Not yet defined
      |       ^~~~~~~~~~~
/usr/bin/ld: /tmp/ccmLGENX.o: in function ```main':
example.c:(.text+0x2e): undefined reference to ```parseOption'
collect2: error: ld returned 1 exit status
```

As early as possible, show the reader an example they can play with. Build on that foundation while keeping the code in a working state.

GOOD: *Show an example the user can build without skipping ahead.*

> ```
> // example.c
> ```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 256

void parseOption(char *line, char *key, char *value) {
    // Fake the parsing part for now.
    strncpy(key, "not implemented", MAX_LINE_LENGTH - 1);
    strncpy(value, "not implemented", MAX_LINE_LENGTH - 1);
}

int main() {
    char line[MAX_LINE_LENGTH];
    char key[MAX_LINE_LENGTH];
    char value[MAX_LINE_LENGTH];

    while (fgets(line, sizeof(line), stdin)) {
        parseOption(line, key, value);
        printf("Key: '%s', Value: '%s'\n", key, value);
    }

    return 0;
}
```

Now, I test the program to see that it runs:

```
$ gcc example.c -o example && \
    printf 'volume:25\npitch:37' | \
    ./example
Key: 'not implemented', Value: 'not implemented'
Key: 'not implemented', Value: 'not implemented'
```

The code fakes parsing options for now, but the dummy code confirms that everything else is working.

Keeping your code in a working state gives the reader confidence that they're following along correctly. It frees them from the worry that they'll waste time later retracing their steps to find some minor error.

4.11. Teach one thing

A good tutorial should explain one thing and explain it well.

A common mistake is to claim a tutorial is about a particular topic, and then bury the lesson in a hodgepodge of unrelated technologies.

BAD: *Teach several unrelated ideas simultaneously.*

> In this tutorial, I'll show you how to add client-side search to your Hugo blog so that readers can do instant, full-text search of all your blog posts, even on spotty mobile connections.
>
> But that's not all!
>
> While I'm showing full-text search, I'll simultaneously demonstrate how you can use browser local storage to store your user's search history and then use an expensive AI service to infer whether the user prefers your website's dark mode or light mode UI theme.

In the example above, the tutorial starts by promising something many readers want: full-text search of a blog.

Immediately after promising full-text search, the tutorial layers in a grab bag of unrelated ideas. Now, anyone interested in full-text search has to untangle the search concepts from everything else.

People come to a tutorial because they want to learn one new thing. Let them learn that one thing in isolation.

GOOD: *Focus your tutorial on a single new concept.*

> In this tutorial, I'll show you how to add client-side search to your Hugo blog so that readers can do instant, full-text search of all your blog posts, even on spotty mobile connections.
>
> That's the only thing I'll demonstrate in this tutorial.

4.11.1. If you have to stack technologies, wait until the end

Sometimes, a tutorial has to combine technologies.

For example, the PHP web programming language doesn't have a production-grade web server built-in. To demonstrate how to deploy a PHP app to the web, you'd have to choose a server like Apache, nginx, or Microsoft IIS. No matter which server technology you choose, you alienate readers who prefer a different web server.

If you have to combine concepts, defer it to the end. If you're teaching PHP, take the tutorial as far as you can go using PHP's development server. If you show how to deploy the PHP app in production using nginx, push those steps to the end so everyone who prefers a different web server can follow everything in your tutorial until the web server portion.

4.12. Don't try to look pretty

Here's an excerpt from an article I read recently. Can you guess what type of tutorial it was?

```
<div class="flex flex-row mb-4 overflow-hidden bg-white">
  <div class="flex flex-col w-full p-6 text-light-gray-500">
    <div class="flex justify-between mb-3">
      <span class="uppercase">{{ title }}</span>
    </div>
    <slot></slot>
  </div>
</div>
```

If you guessed that I was reading a tutorial about a CSS framework, you'd be wrong.

The above snippet was from a tutorial about using the `<slot>` element in the Vue web framework. So, why was half the code just CSS classes? The author added them to make their example look pretty.

Here's the same snippet as above, reduced to the code necessary to convey the concept:

GOOD: *Keep demo UIs simple and style-agnostic.*

```
<div class="card">
  <p class="card-title">{{ title }}</p>
  <slot></slot>
</div>
```

The simplified code doesn't generate a beautiful browser-friendly card, but who cares? It sets a clear foundation to explain the `<slot>` element without distracting you with unrelated technology.

Readers don't care if your toy application looks beautiful. They want a tutorial that makes new concepts obvious.

4.13. Minimize dependencies

Every tutorial has dependencies. At the very least, the reader needs an operating system, but they likely also need a particular compiler, library, or framework to follow your examples.

Every dependency pushes work onto the reader. They need to figure out how to install and configure it on their system, which reduces their chances of completing your tutorial.

Make your tutorial easy on the reader by minimizing the number of dependencies it requires.

BAD: *Surprise the reader with dependencies that are hard to install.*

> We're at step 12 of this tutorial, so it's time to install a bunch of annoying packages I didn't mention earlier:
>
> - ffmpeg, compiled with the libpita extension (precompiled binaries are not available)
> - A special fork of Node.js that my friend Slippery Pete published in 2010 (you'll need Ubuntu 6.06 to compile it)
> - Perl 4

The most common and frivolous dependencies I see are date parsing libraries. Have you seen instructions like this?

BAD: *Add third-party dependencies to solve trivial problems.*

> The CSV file contains dates in `YYYY-MM-DD` format. To parse it, install this 400 MB library designed to parse any date string in any format, language, and locale.

You never need a whole third-party library to parse a simple date string in example code. At worst, you can parse it yourself with five lines of code.

Beyond making your guide harder to follow, each dependency also decreases your tutorial's lifespan. In a month, the external library might push an update that breaks your code. Or the publisher could unpublish the library, and now your tutorial is useless.

You can't always eliminate dependencies, so use them strategically. If your tutorial resizes an image, go ahead and use a third-party image library instead of reimplementing JPEG decoding from scratch. But if you can save yourself a dependency with less than 20 lines of code, it's almost always better to keep your tutorial lean.

4.13.1. Pin your dependencies to specific versions

Be explicit about which versions of tools and libraries you use in your tutorial. Libraries publish updates that break backward compatibility, so make sure the reader knows which version you confirmed as working.

BAD: *Use poorly defined dependencies.*

> Install a stable version of Node.js.

GOOD: *Declare explicit versions for your dependencies.*

> Install Node.js 22.x. I tested this on Node.js v22.12.0 (LTS).

4.14. Specify filenames clearly

My biggest pet peeve in a tutorial is when it casually instructs me to "add this line to your configuration file."

Which configuration file? Where?

BAD: *Give vague instructions about how to edit a file.*

> To enable tree-shaking, add this setting to your config file:
>
> ```
> optimization: {
>   usedExports: true,
>   minimize: true
> }
> ```

If the reader needs to edit a file, give them the full path to the file, and show them exactly which line to edit.

There are plenty of ways to communicate the filename: in a code comment, in a heading, or even in the preceding paragraph. Anything works as long as it unambiguously shows the user where to make the change.

GOOD: *Be specific about which file to edit and where to place changes.*

> To enable tree-shaking, add the following `optimization` setting to your Webpack configuration file under `module.exports`:
>
> ```
> // frontend/webpack.config.js
>
> module.exports = {
>   mode: "production",
>   entry: "./index.js",
>   output: {
>     filename: "bundle.js",
>   },
>   // Enable tree-shaking to remove unused code.
>   optimization: {
>     usedExports: true,
>     minimize: true,
>   },
> };
> ```

4.15. Use consistent, descriptive headings

Most readers skim a tutorial before they decide to read it in detail. Skimming helps the reader assess whether the tutorial will deliver what they need and how difficult it will be to follow.

If you omit headings, your tutorial will intimidate the reader with a giant wall of text.

Instead, use headings to structure your tutorial. A 25-step tutorial feels friendlier if you structure it as a five-step tutorial in which each step has four to six substeps.

4.15.1. Write clear headings

It's not enough to stick a few headings between long stretches of text.

Think about the wording of the headings so that they communicate as much as possible without sacrificing brevity.

Which of these tutorials would you rather read?

> 1. Go
> 2. Installation
> 3. Hello, world!
> 4. Deployment

Or this?

1. Why Choose Go?
2. Install Go 1.23
3. Create a Basic "Hello, World" Go App
4. Deploy Your App to the Web

The second example communicates more information to the reader and helps them decide if this is the right tutorial for them.

4.15.2. Make your headings consistent

Before you publish your tutorial, review your headings for consistency.

BAD: *Use inconsistent headings.*

> 1. How I Installed Go 1.23

2. Step 2: Your First App
3. How I package Go apps
4. Part D: How you'll deploy your App

Checking headings for consistency

- Casing
  - Do your headings use title casing or sentence casing?
- Point of view
  - Are the steps presented as "I did X," "You do X," or neutral?
- Verb tense
  - Are you using present tense, past tense, or future tense?

4.15.3. Create a logical structure with your headings

Ensure that your headings reflect a logical structure in your tutorial.

I often see tutorials where the headings create a nonsensical structure.

BAD: *Use illogical heading structure.*

1. Why Go?
   a. The history of nginx
   b. Configuring nginx for local access
2. Creating your First Go app
   a. Why Go is better than Perl
3. Serve a basic page

In the example above, the heading "Why Go?" has a subheading of "The history of nginx," even though nginx's history isn't a logical subtopic of Go.

4.16. Demonstrate that your solution works

If your tutorial teaches the reader how to install a tool or integrate multiple components, show how to use the result.

BAD: *Show installation but nothing else.*

> Finally, run this command to enable the nginx service:
>
> ```
> sudo systemctl enable nginx
> ```
>
> Congratulations! You're done!
>
> I assume that you know how to do everything from here, so I offer no further guidance.

If you explain how to install something, use the result to show the reader how it works.

Your example can be as simple as printing out the version string. Just show how to use the tool for *something* so that the reader knows whether or not the tutorial worked.

GOOD: *Show the reader how to interact with the tool.*

> Finally, run this command to enable the nginx service:
>
> ```
> sudo systemctl enable nginx
> ```
>
> Next, visit this URL in your browser:
>
> - http://localhost/
>
> If everything worked, you should see the default nginx success page.
>
> 
>
>
> *Figure 11. The nginx success page*
>
> In the following sections, I'll show you how to replace nginx's default webpage and configure nginx's settings for your needs.

4.17. Link to a complete example

Even if you're diligent about keeping the reader oriented throughout the tutorial, it still helps to show how everything fits together.

Link the reader to a code repository that contains all the code you demonstrated in your tutorial.

Ideally, the repository should run against a continuous integration system such as CircleCI or GitHub Actions to demonstrate that your example builds in a fresh environment.

4.17.1. Bonus: Show the complete code at each stage

I like to split my repository into git branches so that the reader can see the complete state of the project at every step of the tutorial, not just the final result.

For example, in my tutorial, "Using Nix to Fuzz Test a PDF Parser," I show the reader the earliest buildable version of the repository in its own branch:

- https://gitlab.com/mtlynch/fuzz-xpdf/-/tree/01-compile-xpdf

At the end of the tutorial, I link to the final result:

- https://gitlab.com/mtlynch/fuzz-xpdf

If your tutorial involves files that are too large to show for each change, link to branches to show how the pieces of your tutorial fit together.

Chapter 5. Write Useful Commit Messages

Effective commit messages simplify the code review process and aid long-term code maintenance. Unfortunately, commit messages don't get much respect, so the world is littered with useless messages like `Fix bug` or `Update UI`.

There's no widespread agreement about what makes a good commit message or why it's useful, but there are several techniques I've found to be useful from personal experience.

5.1. An example of a useful commit message

Here's an example of a useful, effective commit message:

```
Delete comments for a post when the user deletes the post

In change `abcd123`, we enabled users to leave comments on a post, which we store
in the `post_comments` table.

The problem was that when a user deletes their post, we don't delete the
associated comments from the `post_comments` table. The orphaned rows have no
practical use, but they're occupying space in our database and reducing
performance.

This change ensures that we always atomically delete a post's comments at the
same time we delete the post itself by making the following changes to our
database code:

- Updates the `DeletePost` function to also delete comments in the same database
transaction.
- Adds a database migration to delete orphaned rows we added to `post_comments`
prior to this change.
- Adds a `FOREIGN KEY` constraint to the `post_comments` table so that we'll hit
a database error if we ever accidentally delete a post without deleting its
associated comments.

Fixes #1234
```

This commit message is helpful for a few reasons:

- It presents the most important information early.

- It explains the motivation and effect of the change rather than just summarizing implementation details.
- It's succinct and excludes useless noise.
- It cross-references related bugs and commit hashes.

5.2. What's the point of a commit message?

A commit message serves several roles, which I've listed below from most important to least important:

5.2.1. Helps your code reviewer approach the change

When you send your code out for peer review, the commit message is the first thing your reviewer sees.

The code review is the most important scenario for a commit message, as effective communication at the review stage can prevent bugs or maintenance pitfalls before your code reaches production.

5.2.2. Communicates changes to teammates, downstream clients, and end-users

Beyond your reviewer, other people on your team want to understand if your changes impact their work.

If you're working on an open-source codebase or a project with downstream clients, your commit messages also inform your clients and end-users about how your change impacts them.

5.2.3. Facilitates future bug investigations

After you merge your change, your code will live in the codebase for years or maybe even decades. Developers frequently review the commit history to diagnose bugs and to understand the software, so a clear commit message speeds their investigations.

5.2.4. Provides information to development tools

Many development tools scrape information from commit messages to automate software development chores, such as cross-referencing bugs or generating release notes.

5.3. Organizing information in a commit message

5.3.1. Put the most important information first

In a long commit message, most readers don't want to read the entire thing, so put the most important information at the beginning. This allows the reader to stop reading as soon as they reach the information that's relevant to them.

Journalists call this writing style the inverted pyramid. A good news report begins with the details that readers care about most. As the article progresses, the focus shifts toward details that are relevant only to the most interested readers.

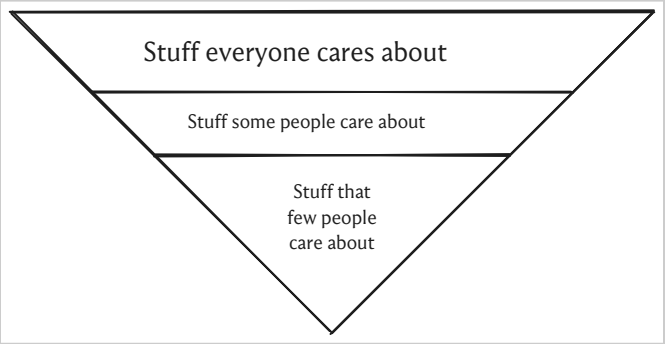


Figure 12. Journalists structure news reports in an inverted pyramid, where the information relevant to the most people is at the top.

5.3.2. Use headings to structure long commit messages

Headings create structure in a long commit message, making it easier for the reader to find information relevant to them:

GOOD: *Use headings to structure long commit messages.*

> **Respond with HTTP 400 if book title contains HTML tags**
>
> With this change, we completely reject requests where the book title contains any HTML tags.
>
> **Background**
>
> We have never allowed HTML in book titles, and our documentation says titles must contain only alphanumeric characters. Prior to this change, we tolerated clients embedding HTML in their book titles because we'd strip out the HTML server-side.
>
> **Motivation**
>
> In bug `#1234`, a user abused our sanitization so that sanitizing the title would actually

> create a `<script>` tag that the attacker used to inject malicious JavaScript. We fixed that in `abcd123`, but then a few months later, in `#4321`, an attacker found a way to inject code through the `onload` attribute.
>
> There is no valid reason for a client to include HTML tags in the title, so if we're seeing them, it's either a misbehaving client or a malicious user.
>
> **Alternative 1 - CSP: Poor library compatibility**
>
> The other way we can handle this is with Content Security Policy (CSP), which would prevent inline JavaScript. The problem is that it would break too many of our existing libraries.
>
> **Alternative 2 - Output encoding: Error-prone**
>
> We could also rely on output encoding so that we always render the HTML as text characters as opposed to HTML. We'd have to get that right 100% of the time we display the title or strings that include the title, so it's too risky.

5.4. What should the commit message include?

For a simple change, a one-line commit message could be sufficient. The more complex the change, the more detail the commit message needs.

The following is a mostly-exhaustive list of details that could be useful in a commit message.

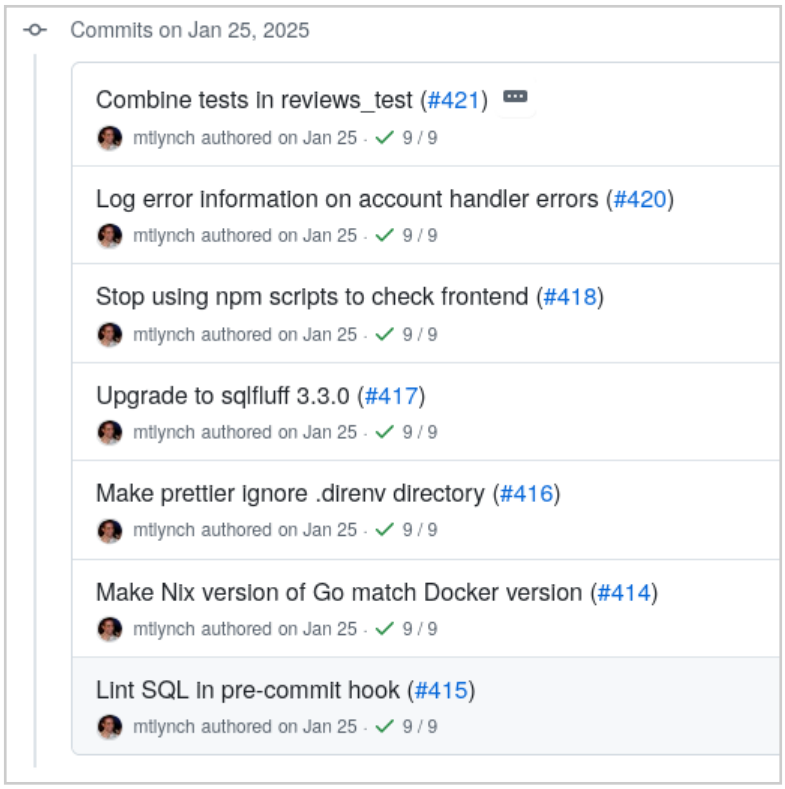
5.4.1. A descriptive title

The first line is the most important part of the commit message because it's what appears in the commit history of most git UIs.

When you print the commit summary using `git log --oneline`, it prints the first line of each commit message:

```
$ git log --oneline
fd8902a (HEAD) Combine tests in reviews_test (#421)
32dbf9a Log error information on account handler errors (#420)
dea3e7a Stop using npm scripts to check frontend (#418)
20ec3c6 Upgrade to sqlfluff 3.3.0 (#417)
4383920 Make prettier ignore .direnv directory (#416)
cacf31b Make Nix version of Go match Docker version (#414)
c6489bf Lint SQL in pre-commit hook (#415)
```

GitHub and other git UIs also show the first line of each commit message prominently in the change history:



Your teammates and users can't read every line of every change to the codebase, so the title is the primary way that you communicate whether the change is relevant to them.

The title should describe the effect of the change rather than how you implemented it. That is, the "what" rather than the "why" or "how."

In a change where you add a mutex to prevent a concurrency bug, this would be a poor title:

BAD: *Use the title to highlight implementation details.*

> Add a mutex to guard the database handle

A better title explains the effect of the change. What's different about the application now that there's a mutex?

GOOD: *Use the title to explain the effect of the change.*

> Prevent database corruption during simultaneous sign-ups

5.4.2. A summary of how the change impacts clients and end-users

Usually, you can't capture all of the relevant details about a change in the title alone, so the subsequent lines of the commit message should fill the gaps.

Keep in mind that, for some audiences, their goal is to understand how the change impacts them without having to read the code itself. It could be because they're an end-user who can't understand software code, or they might just be a busy developer who doesn't have time to read every diff.

Ensure that the commit message explains the relevant details of the change even for someone who won't read the diff itself.

5.4.3. The motivation for the change

After you communicate what the change does, the most important thing to communicate is "why?"

Why are you making this change? How does your change align with the team's goals? What considerations led you to this particular solution?

Consider this example:

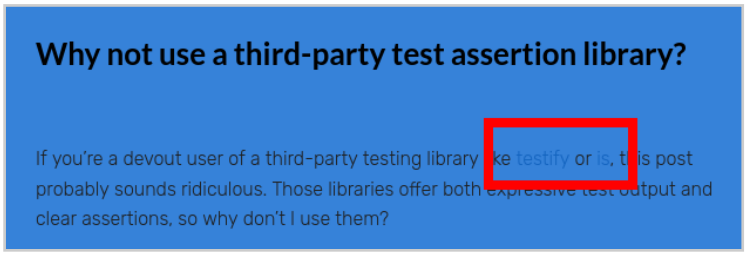
BAD: *Omit the motivation for the change.*

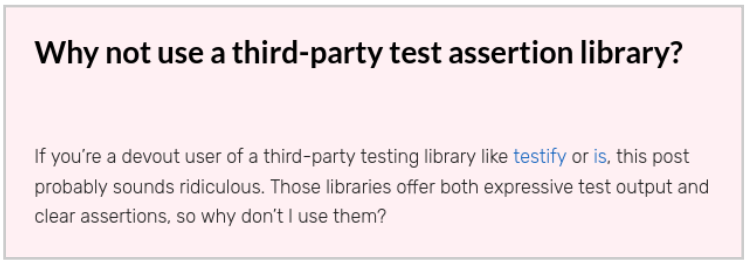
> **Change background from blue to pink**
>
> This updates the CSS so that the application's default background is pink, when previously it was blue.

If a year from now, your teammate wonders why the background is pink, they'd read the above commit message and get zero useful information.

Instead, explain the motivation for the change in the commit message:

GOOD: *Explain the motivation and constraints that influenced the change.*

> **Change background from blue to pink**
>
> Our current blue background makes links difficult to see:
>
> 
>
>
> This changes our background to pink because I think that matches our app's personality, and it makes the links more legible:



The above explanation makes the motivation and reasoning clear. If another developer wants to change the background in the future, they'll understand the constraints that guided the decision to make the background pink.

Sometimes, a change's motivation depends on future plans. You may have a grand vision for how a change is just step one of a beautiful new architecture, but your teammate can't read your mind to see it.

When you send a commit out for review, use the commit message to communicate how the change fits into any larger designs you have in mind.

5.4.4. Breaking changes

If downstream clients have to rewrite code or change their workflows as a result of the change, the commit message should explain what's changing and how clients should handle it.

Use recognizable conventions to make it easy for clients to identify breaking changes. Ensure that your process for creating release announcements scans the commit history to surface details about breaking changes.

GOOD: *Call out breaking changes with a recognizable convention.*

> **Require a signed hash on all requests**
>
> To prevent malicious clients from abusing the server, we're now requiring signed hashes on all requests so that the server can authenticate them as valid.
>
> **Breaking change**
>
> Clients using v2 libraries will no longer be able to access the server. They will need to switch to a v3 or later client implementation.

5.4.5. External references

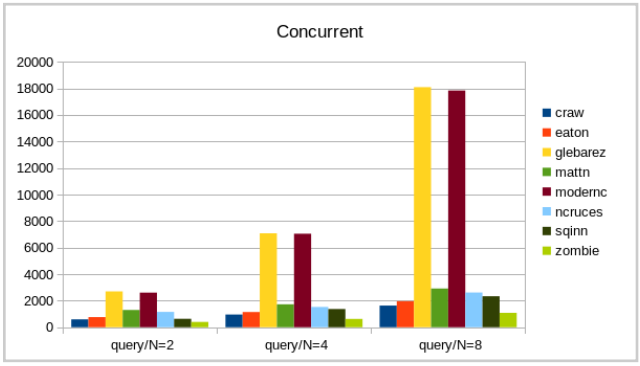
Link to any external documentation or blog posts that influenced your design or implementation choices.

You shouldn't dump your entire browsing history into the commit message (for many reasons), but you should link to the non-obvious resources that will help your teammates understand your thinking.

BAD: *Link to API documentation your teammates can find trivially.*

> This change calls the `database/sql` library, which is documented on the Go docs site:
>
> - https://pkg.go.dev/database/sql

GOOD: *Link to a resource that inspired your choices.*

> I chose the `zombiezen/go-sqlite` SQLite driver, as it outperforms other implementations in high-concurrency scenarios:
>
> 
>
>
> https://github.com/cvilsmeier/go-sqlite-bench/blob/4df8bfd25ea4a0b8fc9460104e7ffb1f6d20cc1a/README.md#concurrent

5.4.6. Justifications for new dependencies

If a change adds a new third-party dependency, flag it in the commit message, and explain why you're adding it.

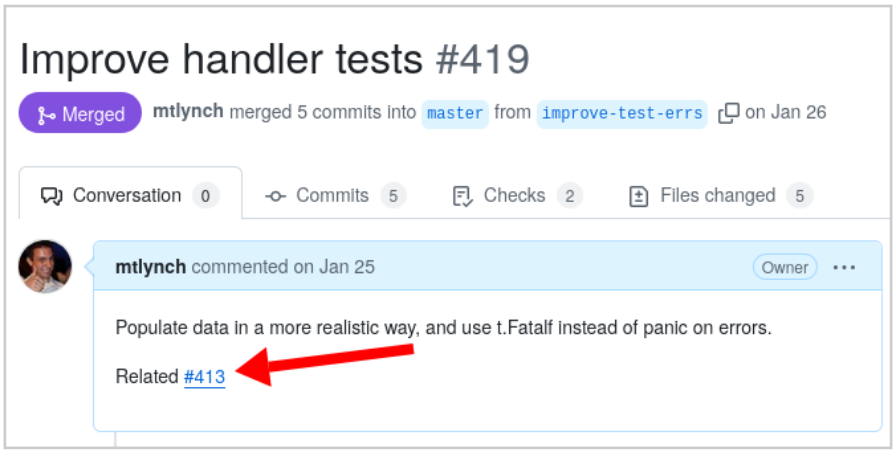
Dependencies are a long-term maintenance burden and a frequent source of bugs. It's helpful for your teammates to understand why you're adding the dependency and how you selected it.

If you're a JavaScript developer, ignore this note, and continue adding 12 new npm package dependencies in each of your commits.

5.4.7. Cross-references to issues or other changes

On most git hosting platforms like GitHub, GitLab, and Codeberg, mentioning an issue by ID like `#1234` automatically creates a cross-reference between the commit and the issue. Similarly, mentioning another change by pull request ID or commit hash creates a cross-

reference.



There are also auto-closing keywords like `Fixes` or `Resolves`. When a commit message includes text like `Fixes #1234`, GitHub, GitLab, and Codeberg all know to auto-close issue #1234 when you merge the change.

Creating cross-references to related issues and other changes helps teammates and future maintainers understand the context around the change.

5.4.8. Summaries of bugs or external references

When you link to a bug or an external reference, don't just write `Fixes #1234` and expect readers to read the entire ticket discussion, especially when the bug has a long, complex history. Summarize the relevant details for the reader.

BAD: *Force the reader to dig through a complex external reference.*

> **Show error instead of blank screen after login**
>
> Fixes #1234

GOOD: *Summarize the details of a bug that are relevant to its fix.*

> **Warn user if they have a malicious Firefox extension**
>
> This change adds a check for the BreakRandomWebsites Firefox extension and warns the user on page load when we detect it.
>
> **Background**
>
> In #1234, a user reported that they got a blank screen after logging in on Firefox, but it worked fine on Safari.
>
> It turned out that the user had the BreakRandomWebsites Firefox extension installed,

> which breaks our app, so we need a way to surface this information to the user more obviously.
>
> Fixes #1234

5.4.9. Testing instructions

Ideally, automated tests should exercise your changes, but if those don't exist, explain to your reviewer how to test your code.

GOOD: *Explain to your reviewer how to test your changes.*

> To test the new behavior:
>
> 1. Populate the testing database with 400 users: `./scripts/populate-store --count 400`
> 2. Run the server in dev mode: `PORT=9000 TESTING=1 ./server`
> 3. Open the login page in the browser: `http://localhost:9000/login`
> 4. Log in as user `admin` / `admin`
> 5. Under "Metrics" click "Delete All"
> 6. Reload to see the server automatically repopulate the metrics tables

5.4.10. Testing limitations

The default expectation is that you tested your commit to your team's normal standards before sending it for review or merging it in the codebase.

In some cases, it's impractical to test a change in all the relevant scenarios. If that happens, disclose what scenarios you haven't tested. It will help your reviewer assess the risk of the change, and it will provide an answer when future maintainers ask, "Did this code *ever* work correctly?"

GOOD: *Explain testing limitations.*

> I don't have a bare-metal RISC-V machine to test this on, but I emulated RISC-V on my AMD64 dev system using qemu, and it worked as expected.
>
> ```
> ./dev-scripts/run-tests --qemu-emulate riscv64
> ```

5.4.11. What you learned

Use the commit message to capture what you learned while implementing the change.

You'll be glad you wrote it down while it's fresh in your mind, and your teammates will be thankful that you spared them all from tracking down the same information.

You don't have to be an expert on whatever you're explaining — admit freely what you don't understand. You'll be surprised at how often your explanations help teammates you assumed knew everything.

BAD: *Force the reader to rediscover everything you had to learn.*

> **Fix a bash bug in the benchmarking script**
>
> This fixes a bug in the benchmarking script related to pipe characters. See the pipelines section of the bash manual for more details:
>
> https://www.gnu.org/software/bash/manual/html_node/Pipelines.html

GOOD: *Explain what you learned from making a change.*

> **Measure execution time more accurately in the benchmarking script**
>
> This fixes a bug in our benchmarking script that caused us to underestimate `evm's performance on our benchmarks.
>
> **evm** has an internal performance timer that starts as soon as the program begins, but our benchmarking script had a bug that started `evm's timer before it could begin working, which made our benchmarks higher (worse) than they were in reality.
>
> **How bash pipelines actually work**
>
> I had a mistaken mental model that bash pipelines execute each process serially.
>
> For example, consider this pipeline:
>
> ```
> ./jobA | ./jobB
> ```
>
> I thought that in the pipeline above, **jobA** would run to completion, then the script would start **jobB** with `jobA's standard output as its standard input.
>
> It turns out that what bash actually does is start both **jobA** and **jobB** simultaneously. **jobB** just blocks on input until **jobA** writes to standard output.
>
> In retrospect, this makes total sense because I've seen plenty of pipelines where the

> second process begins processing output before the first process terminates, but I always modeled the pipeline incorrectly in my head.

One neat side effect of documenting your lesson is that the act of explaining it deepends your understanding. Sometimes, explaining your solution makes you realize there's a better alternative you didn't explore or a corner case you forgot to cover.

5.4.12. Alternative solutions you considered

Often, you begin a change with a naïve solution and find some reason that it doesn't work. When that happens, help your reviewer and future readers understand why the obvious solution didn't work.

Ideally, the explanation should live in a comment within the code itself rather than in the commit message. Otherwise, everyone who reads the code will wonder why it's doing the non-obvious thing.

BAD: *Explain gotchas in the commit message that belong in the code itself.*

> I tried deleting the `time.sleep()` call, but that caused a deadlock between the renderer and the scheduler.

If it's a decision that doesn't have a logical place in the code, explain it in the commit message.

GOOD: *Explain failed approaches that the code can't express.*

> I originally tried to use `std.xml.Parser`, but it doesn't include line-level metadata, which we need for printing error messages.

5.4.13. Searchable artifacts

If a change is related to a unique error message, make sure that the text of the error appears in either the commit message or in a bug that the commit cross-references.

Including the error message ensures that if you encounter that error in the future, you can use `git log --grep` or the search function of your git hosting tool to find past work related to the error.

```
$ git log  --pretty=format:"%s%n%n%b" --grep "reading 'parentNode'"
Fix unclosed div in index.html

On about 30% of builds, the build was failing with this error message:

  Cannot read properties of null (reading 'parentNode')
```

```
It turned out to be caused by an unclosed div in one of our files.
```

Beyond error messages, think about other terms that you or a teammate might search to find related commits, such as names of components, projects, or client implementations.

5.4.14. Screenshots or videos

For changes to an app's user interface, videos and screenshots can be helpful, especially if you show the before and after.

Don't spend hours trying to produce a perfect, Oscar-winning screencast — a simple 15-second demo can make a major difference for your teammates trying to understand the change.

A video should supplement the commit message, not replace it. A rambly, five-minute screencast is a poor substitute for a succinct and well-organized commit message.

The drawback of embedded media is that git doesn't support it natively. If you use git through a platform like GitHub or GitLab, it's easy to embed images in pull request descriptions, which you can subsequently convert to be the commit message for the change.

If you don't use a UI for git that supports embedded media, you can still upload your media to permanent storage and link it as a plain URL in your commit message, though that increases the risk of links going dead if the storage location goes offline or moves. In these cases, it makes more sense to share the screenshots or video in a review comment rather than in the commit message.

5.4.15. Rants and stories

Stories and passionate rants about the code are fun and sometimes informative, so feel free to include them, but save them until the end.

If someone's chasing a high-priority bug at 2 AM, they don't want to read 20 paragraphs about your adventure writing this change or your snarky critiques of a bad library.

BAD: *Bury critical information in a long rant.*

> **Tame the wicked beast that is the load balancer**
>
> Gather friends, as I tell you a tale about my glorious fight with the dastardly foe that we know and love: the load balancer.
>
> It all started back in 1983, when I was a freshman at Diltmore College…
>
> \[50 paragraphs later\]

> When I woke up, my tonsils were missing, and cuddled next to me in bed were 15 stray puppies that I had no memory of adopting.
>
> A sensation in my hand caused me to look down. I was holding a piece of paper smeared with what was unmistakably McDonald's barbecue sauce. Looking closer, I realized the sauce spelled out a single word: goggle.
>
> I snapped open my laptop and ran `grep -i goggle ./src`. Everything suddenly made sense.
>
> There was a typo in our config file that was causing us to ping goggle.com rather than google.com, so this change fixes the typo.

Present critical details succinctly at the top of the commit message, and append the rant to make it obviously extra credit reading.

GOOD: *Present important details first and defer rants until the end.*

> **Fix google.com typo in the load balancer's connectivity check**
>
> Due to a typo introduced in `abc1234`, the load balancer was checking for a live HTTP server at `goggle.com` rather than the intended `google.com`.
>
> This fixes the typo so that we send HTTP requests to `google.com` to verify that the load balancer has Internet connectivity.
>
> ***Rant (just for fun)***
>
> Gather friends, as I tell you a tale about my glorious fight with the dastardly foe that we know and love: the load balancer…

5.4.16. Anything you're tempted to explain outside of the commit message

Sometimes, when someone sends me code to review, they write me an accompanying email to explain background information. If I'm particularly unlucky, they'll interrupt me at my desk to give me a brief lecture that I'm supposed to remember while reviewing their code.

In these cases, I gently tell my teammate that we actually already have a great place to share this information with me: the commit message!

Resist the temptation to explain details about your change outside of the commit message or the code itself. If you need to set the stage before a review for your reviewer, do it in the commit message. That way, your reviewer will see it, and so will anyone else who ever needs that information as well.

5.5. What should the commit description leave out?

It's better to err on the side of overcommunicating in a commit message, but there's also value in reducing noise. Look for opportunities to cut unnecessary information wherever possible.

5.5.1. Information that's obvious from the code

Avoid mentioning facts in the commit message that the code makes plainly obvious. Examples include:

- A list of files you edited
- What APIs you called
- Whether it's a large or a small change

The reader will trivially learn those details when they look at the code, so you don't have to spell it out for them.

5.5.2. Critical details about maintaining the code

You should capture crucial details about maintaining the code, but they're too important to bury in the commit message. Future maintainers might not think to check there when making changes.

If your teammates need to know something about the code, put the information in the code itself, ideally with automated checks to prevent anyone from violating the code's assumptons.

BAD: *Bury critical maintenance information in the commit message.*

> In `disk.c` the offset is `32`, but in `file.c`, the offset is `16`.
>
> The 2:1 ratio is critical, so, if we ever change one of these values in the future, we have to maintain the ratio. Otherwise, the server will silently corrupt all user data, including our offsite backups.
>
> I haven't documented this in the code, as I assume all future maintainers will always follow the blame history to this commit message before editing either of the two files.

5.5.3. Short-term discussion

Consider whether the details you're adding to the commit message are useful to keep in the source history forever or if they're just ephemeral discussion that's useful right now.

If the information is only relevant during the code review, add it as a comment in the discussion for the pull request or bug rather than including it in the commit message forever.

BAD: *Include chatter in the commit message.*

> **@mtlynch** - Can you look at this and tell me if I've gone bonkers?

5.5.4. Preview URLs and build artifacts

If your continuous integration system produces preview URLs or build artifacts that are useful to your reviewers, they should be easy for the reviewer to access, but it shouldn't be the author's job to add them to the commit message manually. Invest in tooling that automatically surfaces the information during the code review.

The more toil you add to the commit message, the more people will perceive it as a mechanical chore rather than an opportunity to communicate useful information.

Consider the lifetime of the build artifacts. If they'll only remain available for a few weeks, then links to build artifacts become distracting noise in the commit message. They should live in the discussion channel rather than the commit message itself.

Chapter 6. Write Emails with Less Noise and Better Results

Coming soon.

Chapter 7. How to Write Compelling Software Release Announcements

A release announcement showcases how the user's experience is better today than it was yesterday. That sounds obvious, but most release announcements forget that there's a user at all.

So many release announcements just enumerate new features in a way that's totallly divorced from how real people use the software. The announcement is essentially just a changelog with better writing.

For example, here's a "changelog" style of announcing a new feature:

BAD: *Describe what the dev team did rather than how it benefits the user.*

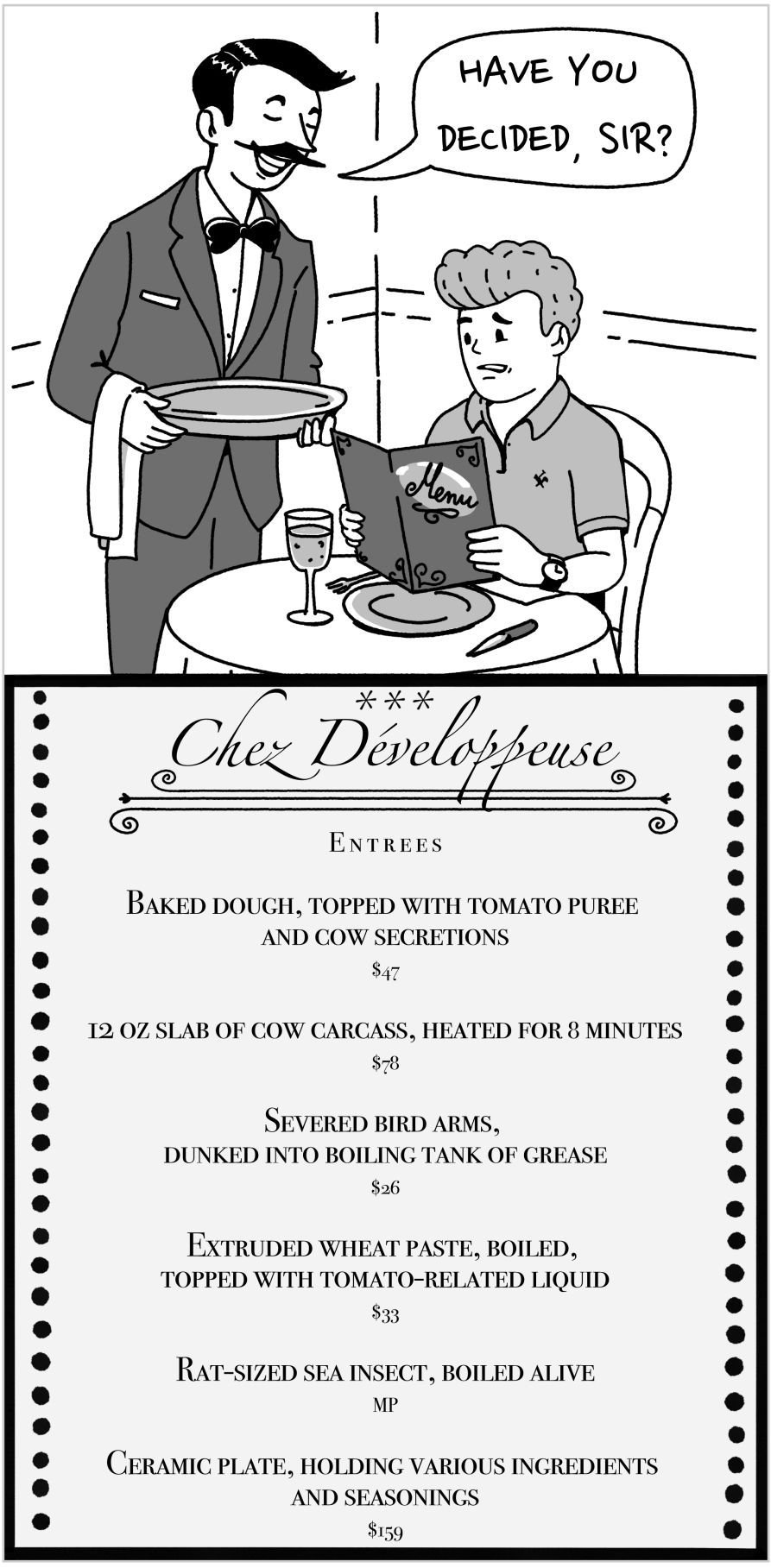
> - Added "repeat" button to the event menu.

Don't just tell the user that there's a new button. Tell the user what they can do with that button.

GOOD: *Describe how changes benefit the user.*

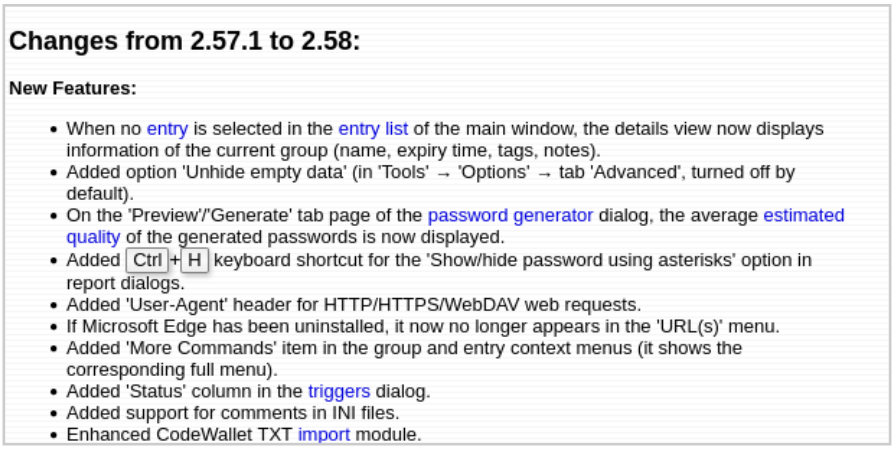
> **Create recurring events automatically**
>
> In previous versions, the only way you could create weekly or monthly events was to duplicate events manually for every occurence. Duplicating events this way was tedious and error-prone.
>
> Our latest release allows you to create recurring events automatically. Click "Options > Repeat", and you'll find options to repeat on any schedule you want. You can even sync with your company's holiday schedule to automatically skip dates that fall on a holiday.

Note that the example doesn't boast about what the software can do. It tells the user what they can do with the software. It speaks directly to the user, describing what happens when "you" create events.


HAVE YOU DECIDED, SIR?
Menu
***
Chez Développeuse
Entrees
Baked dough, topped with tomato puree and cow secretions
$47
12 oz slab of cow carcass, heated for 8 minutes
$78
Severed bird arms, dunked into boiling tank of grease
$26
Extruded wheat paste, boiled, topped with tomato-related liquid
$33
Rat-sized sea insect, boiled alive
MP
Ceramic plate, holding various ingredients and seasonings
$159

7.1. Release notes are not release announcements

Some software companies dump their commit history into a document, add some headings, and call that a release announcement.



Changes from 2.57.1 to 2.58:

New Features:

- When no entry is selected in the entry list of the main window, the details view now displays information of the current group (name, expiry time, tags, notes).
- Added option 'Unhide empty data' (in 'Tools' → 'Options' → tab 'Advanced', turned off by default).
- On the 'Preview'/'Generate' tab page of the password generator dialog, the average estimated quality of the generated passwords is now displayed.
- Added Ctrl+H keyboard shortcut for the 'Show/hide password using asterisks' option in report dialogs.
- Added 'User-Agent' header for HTTP/HTTPS/WebDAV web requests.
- If Microsoft Edge has been uninstalled, it now no longer appears in the 'URL(s)' menu.
- Added 'More Commands' item in the group and entry context menus (it shows the corresponding full menu).
- Added 'Status' column in the triggers dialog.
- Added support for comments in INI files.
- Enhanced CodeWallet TXT import module.

That is not a release announcement. Those are release *notes*.

Release notes are fine and practical, but they're boring.

You can publish release notes as well, but if you want users to feel excited about your newest release, give them something more interesting to read than a sterile changelog.

7.2. What to feature in a release announcement

A release announcement is a summary of the changes that will have the greatest impact on the user's experience. It presents them in a clear, accessible way that focuses on the user.

In contrast to release notes, which aim to be exhaustive, release announcements include only the most impactful changes.

To decide which features to include in the release announcement, consider these questions:

- What can the user do in the latest version that they couldn't before?
- Which workflows became easier?
- Which workflows became faster?

Notice that the questions don't include, "Which feature took the longest to complete" or "What feature contains the cleverest algorithm?"

I've published releases where the flagship feature was a performance improvement that only

took a few hours of dev work. Implementation cost doesn't always correlate with end-user value.

7.3. Call it "faster" not "less slow"

Some bugs are so frustrating and time-consuming to fix that you forget why you're even fixing them in the first place. When it comes time to write the release announcement, you've forgotten how the bug impacted the user experience and just report that it's gone:

BAD: *Focus on what was previously broken.*

> Fixed a thread deadlock that froze the UI for up to two second when creating a new file.

The user didn't know anything about a thread deadlock, but they experienced the symptoms of it. Instead of focusing on what used to be broken, celebrate how the bugfix improves the user's experience:

GOOD: *Focus on the improvement rather than the previous flaw.*

> We sped up new file creation, so you can now create a new file in under 20ms, a 100x speedup from v1.2.

7.4. Briefly introduce your product

In addition to building loyalty with existing users, a release announcement should pique the interest of new, potential users.

If you publish your release announcement to the web, many of the people who read it have never heard of your product at all.

BAD: *Assume every reader is familiar with your product.*

> I'm excited to announce the 1.0 release of OpenVQ9! After 17 years of hard work, this release is a triumph for all OpenVQ9 fans, as well as the loyal developers in the OpenVQ9 ecosystem (all of whom know what OpenVQ9 is, so there's no need to explain it here).

Early in your announcement, include a quick explanation of what your product does. If the reader isn't familiar with your product, what's the least they need to know to understand the rest of your announcement?

GOOD: *Succinctly explain your product to readers unfamiliar with it.*

> I'm excited to announce the 1.0 release of OpenVQ9, the fully open and customizable software for robot vaccum cleaners.

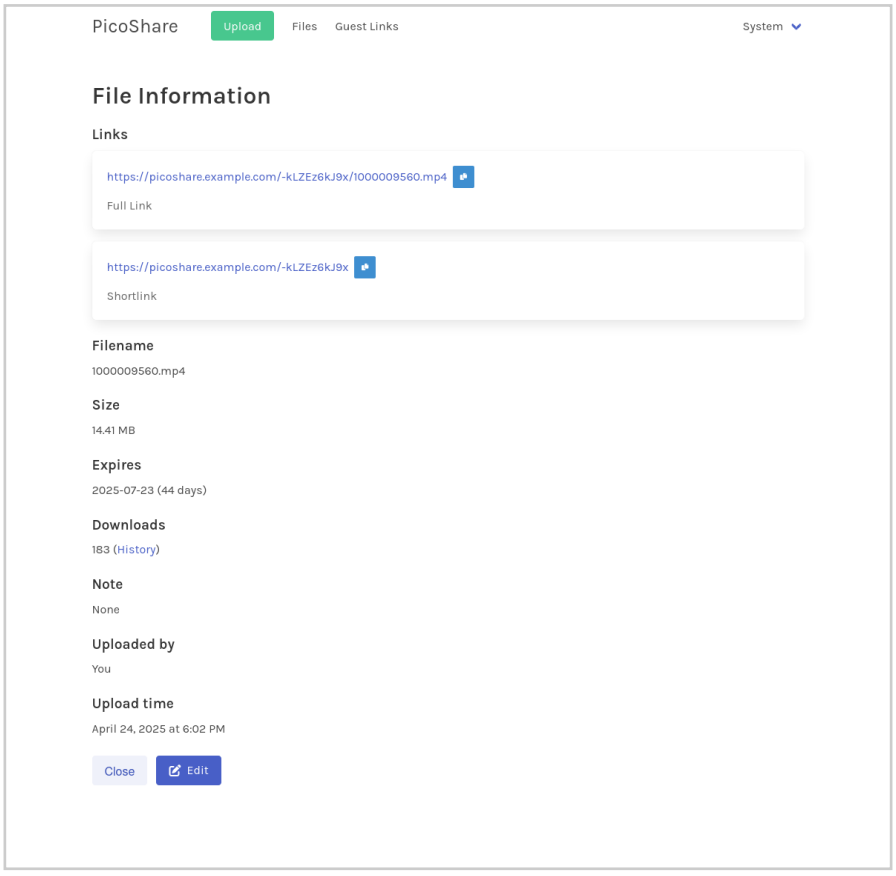
Don't alienate your existing users with a 20-paragraph introduction to a product they already know. A sentence or two will provide context for new users without boring your existing users.

7.5. Make the most of your screenshots

Screenshots liven up a release announcement and make it easier for users to understand new features.

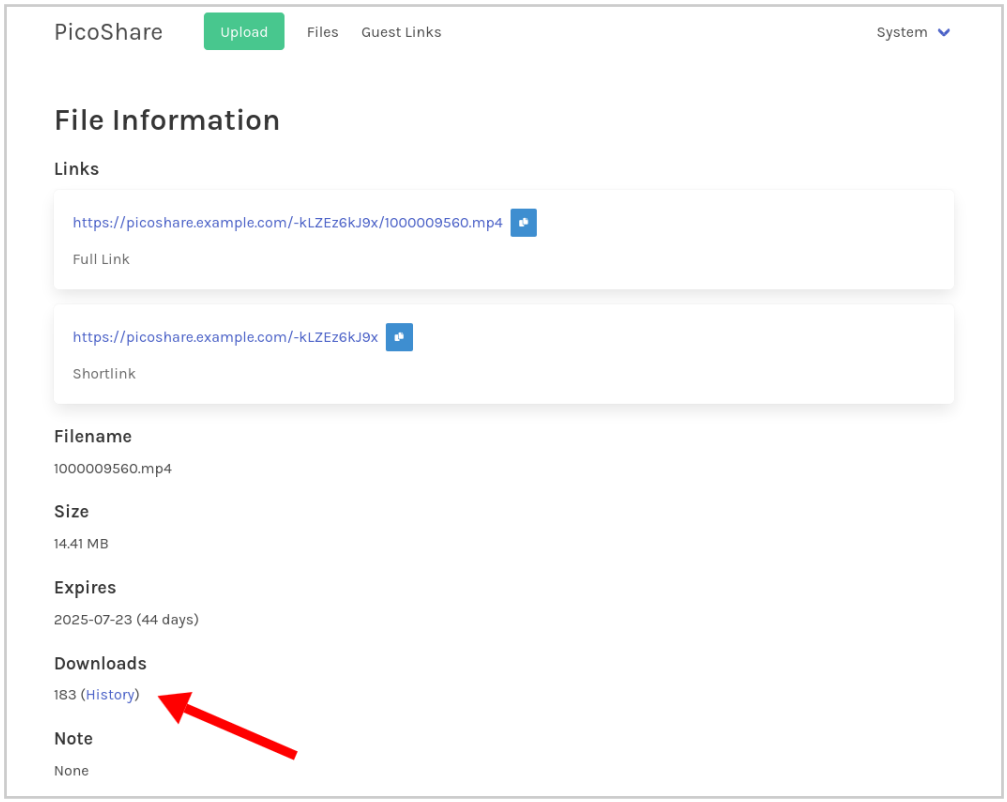
Make screenshots so obvious that the reader recognizes what you want them to see without reading the surrounding text or zooming in. Use cropping, highlighting, or arrows to direct the reader's attention.

BAD: *Make the reader guess about what in the screenshot is notable.*



The above screenshot fails to direct the reader's attention. There are so many UI elements in the screenshot, and none of them have focus.

GOOD: *Make the relevant part of the screenshot obvious.*



The second screenshot draws the reader's focus more clearly by using a tighter crop and adding an arrow to identify the relevant part of the UI.

7.6. Keep animated demos short and sweet

Good news: modern browsers support media formats for playing high-quality screen recordings. We're long past the days of washed-out, pixellated GIFs and clunky YouTube embeds.

- Animated images: WebP and AVIF image formats can show GIF-like animated images, and they enjoy wide browser support.
- Embedded videos: H.264 and WebM-encoded videos play natively in the browser with a simple `<video>` tag. No third-party client library required.

Aim to keep demos under five seconds. 15 seconds should be the maximum. This is both for the sake of respecting the reader's time and your server's bandwidth. Nobody wants to sit through a two-minute video about a dialog box.

7.7. Turn your numbers into graphs

One of my favorite open-source projects recently announced a major performance improvement. Here's how they shared the news with their users:

BAD: *Report numbers to your users with a confusing dump of data.*

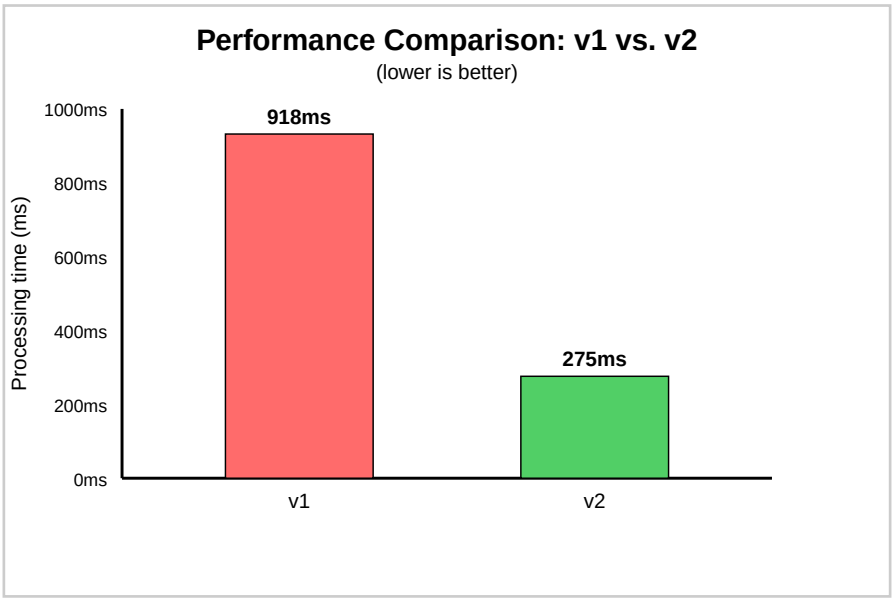
> Here are our performance improvements:

```
Benchmark 1 (6 runs): test-old
  measurement          mean ± σ            min … max      delta
  wall_time           918ms ± 32.8ms     892ms …  984ms   0%
  peak_rss            214MB ±  629KB     213MB …  215MB   0%
  cpu_cycles         4.53G  ± 12.7M     4.52G  … 4.55G    0%
  instructions       8.50G  ± 3.27M     8.50G  … 8.51G    0%
  cache_references    356M  ± 1.52M      355M  …  359M    0%
  cache_misses       75.6M  ±  290K     75.3M  … 76.1M    0%
  branch_misses      42.5M  ± 49.2K     42.4M  … 42.5M    0%
Benchmark 2 (19 runs): test-new
  measurement          mean ± σ            min … max      delta
  wall_time           275ms ± 4.94ms     268ms …  283ms   ⚡- 70.1% ±  1.7%
  peak_rss            137MB ±  677KB     135MB …  138MB   ⚡- 36.2% ±  0.3%
  cpu_cycles         1.57G  ± 9.60M     1.56G  … 1.59G    ⚡- 65.2% ±  0.2%
  instructions       3.21G  ±  126K     3.21G  … 3.21G    ⚡- 62.2% ±  0.0%
  cache_references    112M  ±  758K      110M  …  113M    ⚡- 68.7% ±  0.3%
  cache_misses       10.5M  ±  102K     10.4M  … 10.8M    ⚡- 86.1% ±  0.2%
  branch_misses      9.22M  ± 52.0K     9.14M  … 9.31M    ⚡- 78.3% ±  0.1%
```

If you frequently use the `hyperfine` benchmarking tool, this output format is intelligible to you. To anyone else, it's a barf of confusing numbers. Which of these numbers is supposed to be important? Are these numbers good or bad?

Your users are not performance experts. Instead of dumping a bunch of numbers on their heads, present the information in a graph so that the message is clear and unambiguous.

GOOD: *Translate confusing numbers into a clear graph.*



7.8. Plan your release announcement during development

At my last company, I was in charge of both release planning and release announcements.

I once dedicated an entire release to overhauling our product's self-update feature. The previous logic was so messy that it took five months to replace it, twice the turnaround of our typical release.

When I finally wrote the release announcement, I realized I'd screwed up: nothing in the release benefitted our users. Sure, future releases would be faster and less error-prone, but the user's experience today was no better than it was yesterday.

I awkwardly tried to explain to our users why we made them wait five months for a release that essentially did nothing for them. I promised that they'd benefit from these changes soon, as the new update system would accelerate development (and it did), but I was embarrassed that the release primarily made life better for our developers rather than for our users.

From then on, I always considered the release announcement early in release planning. I still invested in maintenance and refactoring work, but I made sure to always have a few changes in the release that we could showcase in the release announcement.

7.9. No more "various improvements and bugfixes"

A release announcement should never include the phrase, "various improvements and bugfixes." You might as well boast that the team proudly breathed air throughout development and used the latest version of the Internet.

If you can't articulate how a change benefits your users, don't highlight it in your release announcement. Save the exhaustive list of changes for your release notes, but even there, please leave out "various improvements and bugfixes."

7.10. Real-world examples of compelling release announcements

7.10.1. Gleam JavaScript gets 30% faster

The release announcement for Gleam 1.11 puts the flagship feature right in the title: a 30% performance improvement. The title makes it crystal clear that the announcement focuses on what the user cares about. Rather than just throwing numbers at the reader, the author visualizes the improvements with clear graphs.

The Gleam release announcement describes every feature in terms of user impact. Rather than bore the reader with a dry list of changes, the announcement accompanies every change with code snippets showing how each change in the language and toolset has made life better for Gleam's users.

7.10.2. Navigating UI3: Figma's new UI

Figma is a design tool that earned a strong reputation for its relentless focus on user experience. This same focus is evident in their release announcements.

Figma's UI3 release announcement focuses on empowering the user. You won't find anything like, "this button is now here, and we added this dialog." Instead, they tell the user "**You** can now do X to achieve Y." That is, instead of explaining what changed, they showcase how the change benefits the user.

My only criticism is that Figma's release announcement bizarrely uses 8 MB GIFs (yes, multiple) to show 5-second demo animations.

Summary

- Highlight new features and changes that excite the user.
  - The exhaustive accounting of every change belongs in the release notes, not the release announcement.
- Describe changes in terms of what improved rather than what is no longer bad.
- Include a succinct introduction of your product in case the reader hasn't used it yet.
- Use screenshots that make the new feature obvious even if the reader doesn't zoom in or read the surrounding text.
- Include animated demos, but keep them under 15 seconds.
- Prefer graphs to raw numbers.
- Cut out vague descriptions like "various improvements and bugfixes."

Chapter 8. Write Effective Design Documents

Coming soon.

Chapter 9. Make Your Writing Sound Natural

"Read your writing aloud."

For most of my life, I heard that advice and ignored it. It was the same reaction I had when personal finance experts recommended making a formal budget for my monthly expenses. "I can predict what that would feel like, and I don't think it's valuable, so I don't have to try it."

When I finally tried reading my writing aloud, the value was immediately obvious. I caught awkward phrasing and careless errors that I'd overlooked when reading the same thing in my head.

Before I publish anything, I send the draft to my e-reader, leave my office, and read my draft sompelace quiet. I don't know how much difference it makes, but I choose a reading environment that's distinct from where I did my writing. As I read, I scribble notes on the pages to reflect my ideas for edits.

So, just try it once. It will take you 10 minutes to try, and if you're not convinced, you only risked 10 minutes of your life.

As you read, pay attention to your intuition. If you find yourself tripping over your words or saying something that feels awkward, your reader will probably experience the same thing.

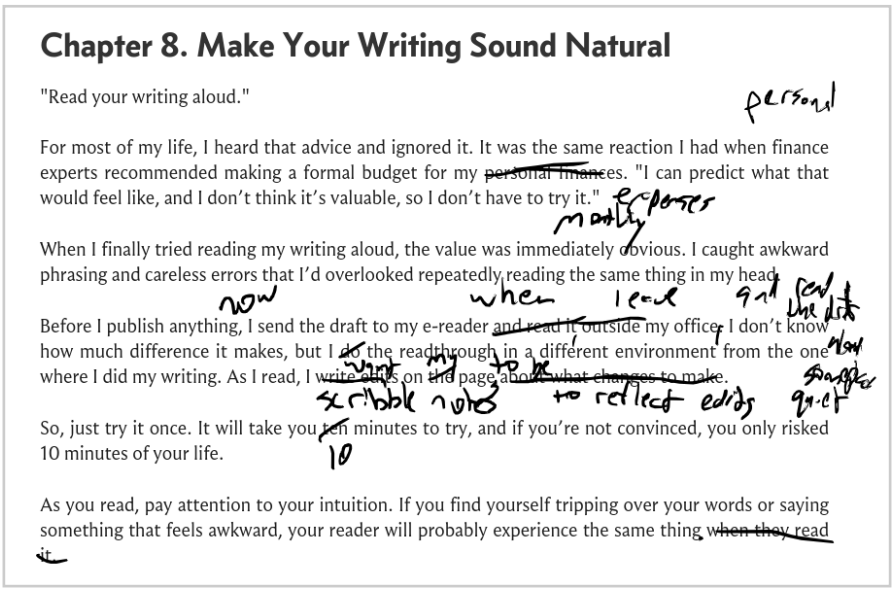


Figure 13. Example of notes I made while reading this chapter aloud.

Red flags to look out for when reading your writing aloud

- Do any sentences cause you to stumble over your words?
- Do you find yourself wanting to say something different from what you wrote?
- Do certain sections make you feel bored or disinterested?

Chapter 10. Fine-Tuning Your Writing

Coming soon.

10.1. Verbs drive the sentence

Coming soon.

10.2. Stay positive: how negative phrasing reduces readability

Coming soon.

10.3. Passive voice considered harmful

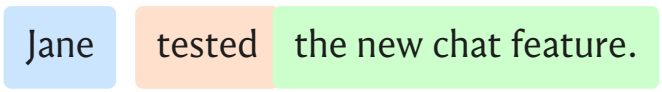
Your high school English teachers probably warned you that passive voice is dangerous and forbidden. Then, when you were an adult, some guy in a leather jacket told you that passive voice is cool and should be used whenever it's desired.

Well, the tide has turned again. If you're a software developer, stop using the passive voice.

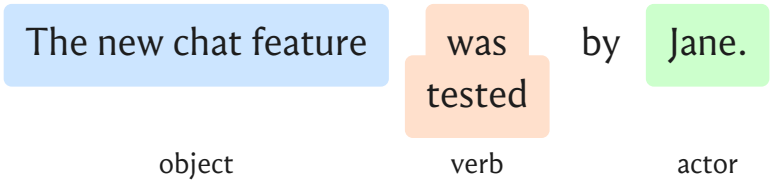
10.3.1. Wait, what's passive voice?

In English, sentences can have one of two structures: passive voice or active voice.

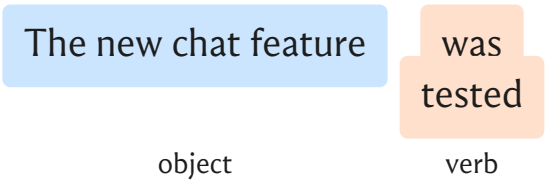
Active voice is when you construct a sentence as "actor → verb → object."



Passive voice is when you construct a sentence as "object → verb → actor."



Passive voice also allows you to construct the sentence as "object → verb" with no actor at all.

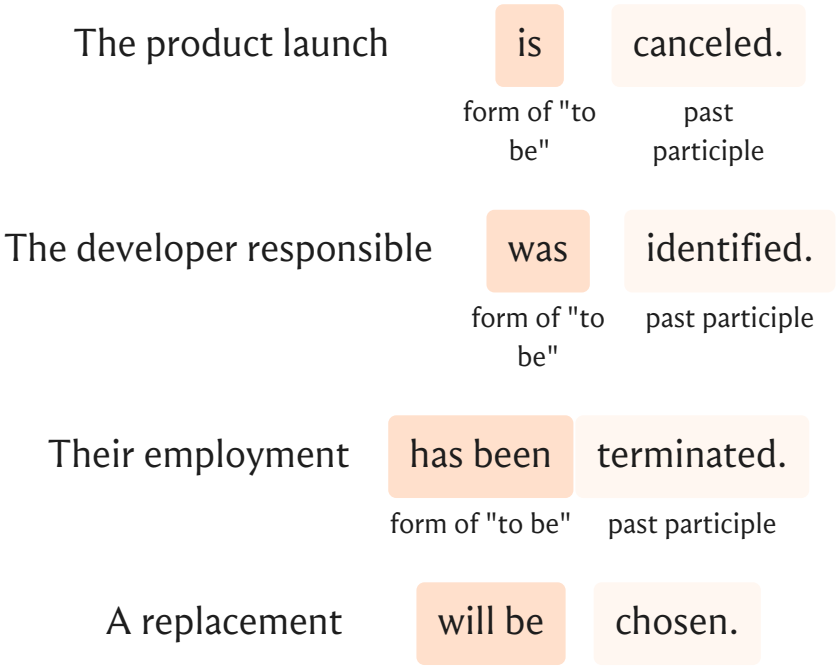


10.3.2. Recognizing passive voice

You can recognize passive voice from two signature features:

1. A form of the verb "to be" (e.g., is, was, will be)
2. The past participle of a verb (e.g., downloaded, tested)

Here are a few examples:



form of "to be" | past participle

10.3.3. What's wrong with passive voice?

If you forgot what passive voice is, you definitely forgot why you're not supposed to use it.

You may have a hazy recollection that passive voice is grammatically incorrect (it's not). Maybe you recall teachers telling you that passive voice is "poor style," but that's vague and hand-wavey.

Regardless of what you remember, there are clear, concrete reasons to avoid the passive voice.

Passive voice omits important information

Passive voice's greatest sin is that it omits critical information.

When I review design documents, I often find lines like this:

BAD: *Use the passive voice to omit key details.*

> For security, the message is signed, and the signature is validated before further processing.

That sentence creates a lot of questions:

- Who signs the message?
- Who validates the message?
- Who performs further processing?

Here's a rewrite of that sentence using the active voice:

GOOD: *Specify which component performs which action.*

> For security, the client signs the message using the client private key. The client then sends the meessage to the orchestration server.
>
> When the orchestration server receives the message, it forwards the message to the authentication service. The authentication service then validates the message's signature.
>
> - If the authentication service successfully validates the message's signature, the orchestration server sends the message to the queuing service.
> - If the authentication service rejects the message, the orchestration server drops the

message and responds to the client with an HTTP 400 error.

- If the authentication service fails to respond within 5 seconds, the orchestration server drops the message and responds to the client with an HTTP 500 error.

"Hang on!" You might be thinking. "That wasn't a rewrite to the active voice. That's completely different text."

But that's the point.

Passive voice draws the reader's attention away from missing details like a magician using misdirection to control the audience's focus. Converting a sentence to active voice makes the missing details obvious, revealing gaps in your design.

Passive voice increases cognitive load for the reader

Most English sentences take the form of actor → verb → object. This structure is familiar to the reader, and it allows them to parse the sentence easily.

Passive voice inverts the typical sentence structure, which adds mental burden for the reader. When they see a verb before seeing its corresponding actor, they effectively have to push it onto their memory stack and then mentally reconfigure the concept into "actor → verb → object" order.

Obviously, writing can sometimes challenge the reader and demand that they carry some cognitive load. But your reader only has finite focus. Reserve their mental energy for when you have a difficult topic to explain, not when you want to use a particular sentence structure.

10.3.4. When is passive voice useful?

Despite all the reasons not to use passive voice, there are a few cases where it's acceptable and, I'll admit: helpful.

As a rule of thumb, whenever you encounter passive voice in your writing, consider a rewrite. But also consider the possibility that, in certain cases, passive voice might be the appropriate choice.

Use passive voice to focus on solutions rather than assigning blame

When you discuss problems with teammates, it's important to focus on solving the problem rather than pointing fingers.

Imagine that you were performing a postmortem on a website outage, and you included this sentence:

BAD: *Use the active voice to focus blame on a teammate.*

> **Michael accidentally shut down** the production server, which caused a three-hour outage on our website.

The phrasing leads the reader to believe that Michael is the problem. If Michael hadn't been so stupid and careless, the outage would never have happened.

Here's a rewrite in the passive voice:

GOOD: *Use the passive voice to shift focus to the problem.*

> The production server **was accidentally shut down**, which caused a three-hour outage on our website.

The passive voice shifts focus away from the individual and leads the reader's attention to the systems that allowed the problem:

- Why should there be an outage if one server shuts down?
- Why is it possible to shut down a critical server by mistake?
- Why does it take three hours to get a critical server back online?

These questions are likely to yield more robust systems than simply blaming the issue on one person's carelessness.

Use passive voice to communicate mistakes tactfully

I find passive voice effective in pointing out someone's mistake in a friendly, professional way, especially if I don't know the person well.

For example, if a customer forgot to include their logs with a bug report, this would be a particularly hostile way of letting them know:

BAD: *Use the active voice to make communication adversarial.*

> It looks like **you ignored my instructions** and **failed to include the log file** in your bug report.

Instead, I would use the passive voice to avoid sounding accusatory:

GOOD: *Use the passive voice to shift focus to the problem.*

> It looks like **the log file was accidentally omitted** from the bug report. Could you try reattaching your log file and emailing it to me?

Use passive voice to exclude irrelevant details

Passive voice omits details, but sometimes that's what you want. Some details are irrelevant

and deserve omission.

GOOD: *Use the passive voice to exclude irrelevant details.*

> I **was fired** from my last job for bringing my pet puma to work.

The sentence above uses passive voice to omit details, but that's okay.

Maybe the person who fired you was named Glenn, and he was a junior HR associate. Maybe he had dreams of being a champion Scrabble player, and his favorite color was purple.

The reader doesn't care about Glenn.

They care about why you have a pet puma, why you thought it was a good idea to bring it to work, and what happened when you did. In that case, it's perfectly acceptable to use passive voice to edit Glenn out of your story.

10.4. Minimize cognitive load for the reader

Coming soon.

10.5. Brevity is performance optimization for writing

Coming soon.

10.6. Eliminating ambiguity and confusion

Coming soon.

Chapter 11. Maintaining Motivation

Coming soon.

11.1. Manage writer's block

Coming soon.

11.2. Using a structured process to stay in flow state

Coming soon.

11.3. Editing: valuable because it's hard

Coming soon.

Chapter 12. Resources to Improve Your Writing

Coming soon.

12.1. Work with a professional editor

Coming soon.

See How I Hired a Freelance Editor for My Blog for some of the ideas that will inform this chapter.

12.2. Work with a professional illustrator

Coming soon.

See How to Hire a Cartoonist to Make Your Blog Less Boring for some of the ideas that will inform this chapter.

12.3. Improve your grammar incrementally

Coming soon.

12.4. Using AI tools

Coming soon.