



WORKING
IN
PUBLIC

THE MAKING AND
MAINTENANCE OF
OPEN SOURCE
SOFTWARE

NADIA EGHBAL



WORKING
IN
PUBLIC

THE MAKING AND
MAINTENANCE OF
OPEN SOURCE
SOFTWARE

NADIA EGHBAL



Ideas for progress
San Francisco, California
press.stripe.com

WORKING IN PUBLIC

THE MAKING AND MAINTENANCE
OF OPEN SOURCE SOFTWARE

BY NADIA EGHLAL

Working in Public: The Making and Maintenance of Open Source Software
© 2020 Nadia Eghbal

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any other information storage and retrieval system, without prior permission in writing from the publisher.

Published in the United States of America by Stripe Press / Stripe Matter Inc.

Stripe Press
Ideas for progress
San Francisco, California
press.stripe.com

ISBN: 978-0-578-67586-2

First Edition
Second printing

Creative Director: Tyler Thompson
Typesetting & Design: Alice Chau

Composed in Favorit & Favorit Mono from ABC Dinamo Foundry

ebook design by [Bright Wing Media](#)

CONTENTS

1. [INTRODUCTION](#)
2. [PART 1: HOW PEOPLE MAKE](#)
 1. [01: GITHUB AS A PLATFORM](#)
 2. [02: THE STRUCTURE OF AN OPEN SOURCE PROJECT](#)
 3. [03: ROLES, INCENTIVES, AND RELATIONSHIPS](#)
3. [PART 2: HOW PEOPLE MAINTAIN](#)
 1. [04: THE WORK REQUIRED BY SOFTWARE](#)
 2. [05: MANAGING THE COSTS OF PRODUCTION](#)
4. [CONCLUSION](#)
5. [CREDITS](#)
6. [NOTES](#)

Landmarks

1. [Cover](#)
2. [Title Page](#)
3. [Copyright Page](#)
4. [Table of Contents](#)
5. [Body Matter](#)

INTRODUCTION

Until recently, information was good, and more information was better. If the free exchange of ideas formed the basis of a flourishing society, then we had a moral imperative to connect more people to one another.

The spirit of openness lasted more than 200 years. We championed the value of literacy and education. We built roads, bridges, and highways that brought together previously isolated communities. We careened toward the new millennium, flushed with the global triumph of Western liberal democracy.

And so, in the late twentieth century, when murmurs of the internet began to hum to an audible roar, they carried with them all the qualities of our harmonious, never-gonna-give-you-up infatuation with the unconstrained spread of knowledge. Tim Berners-Lee, the inventor of the World Wide Web, envisioned “a pool of information . . . which could grow and evolve” in his original proposal to CERN, the European particle physics laboratory that incubated his project.¹ “For this to be possible, the method of storage must not place its own restraints on the information,” he wrote. With his proposal as a blueprint, technologists built a Library of Alexandria bigger than we could have ever conceived in the physical realm, inextricably linking people and their ideas all around the world.

Then we hit a snag. Suddenly, there was too much information. Too many notifications made us want to check them less. Too many social interactions made us want to post online less frequently. Too many emails made us not want to answer. We were, effectively, DDoSing one another: the term for a distributed denial-of-service attack, in which malicious actors overwhelm their target by flooding it with traffic, leaving the victim incapacitated. Our

online public lives became too much to handle, causing many of us to shrink back into our private spheres.

There's a similar story playing out in the world of *open source* software: a term that's nearly synonymous with public collaboration, but whose developers—who write and publish code that anybody can use—often report feeling overwhelmed by the volume of inbound requests.

After interviewing hundreds of open source developers and researching their projects, I summarized this problem in a 2016 report for the Ford Foundation, titled “Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure.”² Then I set out to try to address it.

Much of my stress-testing happened from 2016 to 2018, while I was working to improve the open source developer experience at GitHub, a company that offers hosting for open source projects. As the platform where most open source software is built, GitHub was the perfect place to learn. I met even more developers and worked with more projects, and I had to think pragmatically about solutions. These were the moments, often humbling, when rhetoric squared off with reality.

That many open source developers suffer from a lack of support is indisputable. My inbox was flooded with emails from people involved in open source, eager to share their stories. The hard part was identifying what they actually needed.

Initially, I focused my explorations on a lack of funding. Many open source developers are not directly paid for their work, despite generating trillions of dollars in economic value. In the absence of additional reputational or

financial benefits, maintaining code for general public use quickly becomes an unpaid job you can't quit.

But as I tracked developers' stories over the years, I noticed that money is only part of the problem. Some developers deftly handle demand from their users without any financial compensation. And even paid open source developers seem to go through the same strange behavioral cycle, in a way that seems less common among those who write "proprietary," or private, code for their employers.

The cycle looks something like this: Open source developers write and publish their code in public. They enjoy months, maybe years, in the spotlight. But, eventually, popularity offers diminishing returns. If the value of maintaining code fails to outpace the rewards, many of these developers quietly retreat to the shadows.

A developer who's employed at a private company works primarily with their colleagues. A developer who writes code in public must work with thousands of strangers: literally, anybody with an internet connection who cares to comment on their code. The lack of financial reward is a symptom, perhaps, of misaligned incentives, but this inevitable cycle of churn seems to hint at something deeper.

The default hypothesis today is that, faced with growing demand, an open source "maintainer"—the term used to refer to the primary developer, or developers, of a software project—needs to find more contributors. It's commonly thought that open source software is built by communities, which means that anyone can pitch in, thus distributing the burden of work. On the surface, this appears to be the right solution, especially because it

seems so attainable. If a lone maintainer feels exhausted by their volume of work, they should just bring more developers on board.

There are countless initiatives today aimed at helping more developers contribute to open source projects. These efforts are widely championed as “good for open source,” and they are frequently accomplished by tapping into a public sense of goodwill.

However, in speaking to maintainers privately, I learned that these initiatives frequently cause them to seize with anxiety, because such initiatives often attract low-quality contributions. This creates more work for maintainers—all contributions, after all, must be reviewed before they are accepted. Maintainers frequently lack infrastructure to bring these contributors into a “contributor community”; in many cases, there is no community behind the project at all, only individual efforts.

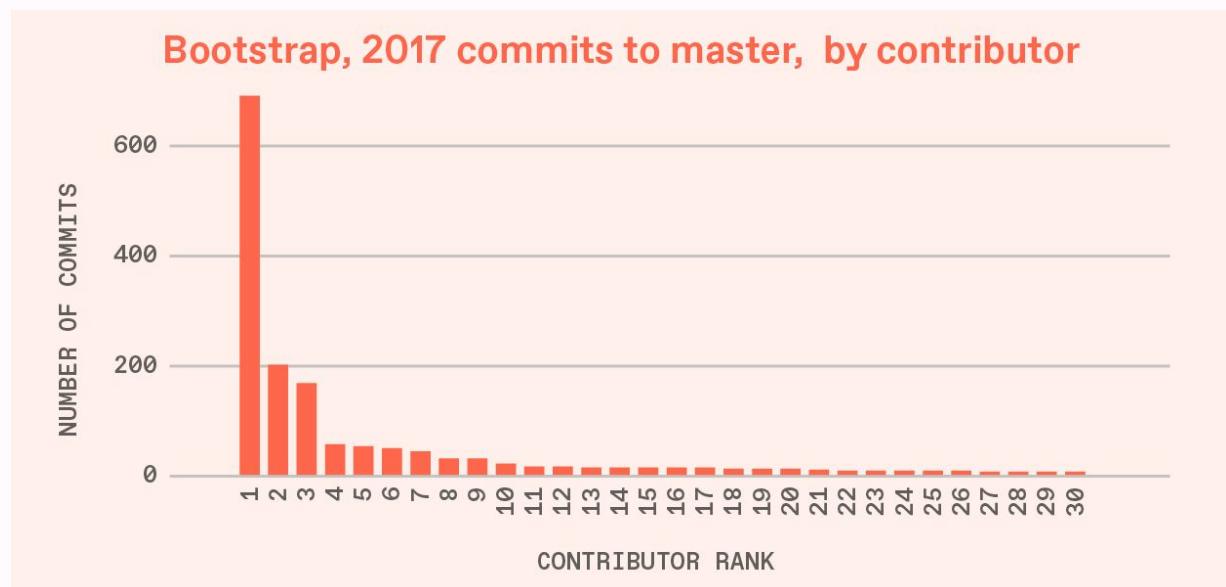
In my conversations with maintainers, I heard them express a genuine conflict between wanting to encourage newcomers to participate in open source and feeling unable to personally take on that work. Maintainers simply don’t have the energy to onboard every person who shows passing interest. Many told me they were frustrated by prior attempts to cater to a revolving door of contributors—sometimes hundreds of them—who didn’t stick around. Maintainers recounted how those who’d expressed interest sometimes disappeared before they’d even submitted their first contribution.

I started to see the problem is not that there’s a dearth of people who want to contribute to an open source project, but rather that there are *too* many contributors—or they’re the wrong kind of contributors. Open source code

is public, but it doesn't have to be participatory: maintainers can buckle under excess demand for their attention.

One study found that, in a sample of 275 popular GitHub projects across various programming languages, nearly half of all contributors only contributed once. These contributors accounted for less than 2% of total *commits*, or overall contributions.³ The regular presence of hundreds or thousands of contributors making few substantial contributions, rather than a smaller group of developers making meaningful contributions, throws the idea of “contributor communities” into question.

I realized there was an enormous disconnect between how we *think* open source works and what is actually happening on the ground today. There is a long, and growing, tail of projects that don't fit the typical model of collaboration. Such projects include Bootstrap, a popular design framework used by an estimated 20% of all websites,⁴ where three developers have authored over 73% of commits.^{*5} Another example is Godot, a software framework for making games, where two developers respond to more than 120 issues opened on their project per week.⁺⁶



Bootstrap's contributors in 2017, plotted by number of commits.

This distribution—where one or a few developers do most of the work, followed by a long tail of casual contributors, and many more passive users—is now the norm, not the exception, in open source. From the perspective of these maintainers, quietly tending to their code on the prairie amidst the tumbleweeds of unanswered issues, the world doesn't look like the utopian vision, embraced by early internet pioneers, of large-scale collaboration among strangers. If anything, it looks like exactly the opposite of what these early advocates predicted. One study found that in more than 85% of the open source projects the researchers examined on GitHub, less than 5% of developers were responsible for over 95% of code and social interactions.⁷ In their report, the study's authors note a puzzling “mismatch between established theory and a widely observed empirical effect.”

While these numbers might seem alarming, it's only because they don't match popular expectations. We assume that open source projects need to grow strong contributor communities in order to survive. There's even a term called the *bus factor*, where project health is measured by the number of developers that would need to get hit by a bus before the project is in trouble.

But this narrative no longer translates to how many open source projects work today. Given that the average software developer easily relies on hundreds of open source projects to write code these days, it's inevitable that they'll only be able to passively consume most of them.

The expectation that open source should be a collaborative effort leaves solo maintainers scratching their heads, wondering if they're doing something wrong when nobody shows up to meaningfully pitch in. But

what if we start from the premise that things are as they should be? I decided to revisit the diagnosis, treating these symptoms as a starting point, instead of applying the only prescription—more participation—that we had.

Code, like any other type of content available online today, is trending toward modularity: a mille-feuille layer cake of little libraries instead of one big, jiggling Jell-O mold. Today, developers publish bits of code online, for public use, as easily as others discover and use them. But just as tweets are easy to read and retweet without context as to who wrote them, code is easy to copy-paste without knowing, or caring, where it came from.

The npm ecosystem—which, according to its parent company, makes up an estimated 97% of the code in modern web applications today—offers some clues to the future.⁸ Npm, which stands for Node Package Manager, is the platform commonly used by JavaScript developers to install and manage *packages*, or libraries. (Libraries are prewritten bits of code that other developers can use, instead of having to write the same code from scratch. The ability to legally reuse other people’s code is what allows software developers to work more quickly and efficiently today.)

In contrast to big, monolithic software projects that give rise to persistent communities, npm packages are designed to be small and modular, which leads to fewer maintainers per project and a transitory relationship with the code they write. Viewed in this light, the lack of contributors today reflects an adaptation to changing environmental circumstances, wherein the relationship between a maintainer, contributors, and users is lighter, more transactional.

When speaking to first-time and casual contributors about their experiences, I found their stories just as enlightening as those of maintainers. Casual

contributors are often aware that they have little context for what is going on behind the scenes of the project. But more importantly, they do not *want* to spend time familiarizing themselves with a project’s goals, roadmap, and contribution process. These developers primarily see themselves as users of the project; they do not think of themselves as part of a “contributor community.”

The role of a maintainer is evolving. Rather than coordinating with a group of developers, these maintainers are defined by the need for *curation*: sifting through the noise of interactions, such as user questions, bug reports, and feature requests, which compete for their attention.

Given limited time and attention, solo maintainers need to balance reactive tasks (community interactions) with proactive ones (writing code).

Maintainers also rely on platforms—namely, GitHub—and tools—such as bots that help with managing issues, notifications, and code quality—to keep up with their work.

The problem facing maintainers today is not how to get more contributors but how to manage a high volume of frequent, low-touch interactions.

These developers aren’t building communities; they’re directing air traffic.

The answer to rising levels of participation isn’t restricting access to code. The number of developers who *use* open source code is growing exponentially. SourceForge, the dominant code-hosting platform back in 2001, had just over 200,000 users at the time.⁹ Today, GitHub, its successor, claims over 40 million registered users, a number that has nearly doubled in just the past two years.¹⁰

That so much information is now freely available at our fingertips is one of the most important wins of the digital age. More importantly: it's not the excessive *consumption* of code but the excessive *participation* from users vying for a maintainer's attention that has made the work untenable for maintainers today.

While this world of lone developers, making things for millions of users, seems like a marked deviation from the story of open source, it's not so different from what's happened online more broadly. Increasingly, our online worlds are being built by individuals, not just communities.[‡] As one Reddit user observes,

If you read reviews on Amazon, you're mostly reading reviews written by people like Grady Harp. If you read Wikipedia, you're mostly reading articles written by people like Justin Knapp. . . . And if you read YouTube comments, you're mostly reading comments written by people like Justin Young. If you consume any content on the Internet, you're mostly consuming content created by people who for some reason spend most of their time and energy creating content on the Internet. And those people clearly differ from the general population in important ways.¹¹

As a developer, if you use the command-line tool cURL, you're using code written by Daniel Stenberg. If you use the command-line interface bash, you're using code maintained by Chet Ramey. If you use npm, you're using packages written by Sindre Sorhus and Substack. If you use Python's packaging tools, you're using code maintained by Donald Stufft.

Like any other creator, these developers create work that is intertwined with, and influenced by, their users, but it's not *collaborative* in the way

that we typically think of online communities.

Rather than the users of forums or Facebook groups, GitHub's open source developers have more in common with solo creators on Twitter, Instagram, YouTube, or Twitch, all of whom must find ways to manage their interactions with a broad and fast-growing audience. Instead of facing one another, as pop culture critic Mark Fisher allegedly once put it, a creator's audience now faces the stage.

Like other creators, open source developers make things for general public consumption. They, too, need to deal with crowded inboxes, manage their limited attention, and lean upon enthusiastic fans for support. Like other creators, open source developers rely heavily upon platforms to distribute their work. As closed economies, these platforms also bear the responsibility of helping creators grow their reputations and capture the value of their efforts.

Early internet activity was characterized by large-scale, distributed online communities: mailing lists, online forums, membership groups. These communities operated as a cluster of villages, each with its own culture, history, and norms.

Social platforms brought all these communities to one place and smashed them together like Play-Doh. In doing so, they inadvertently changed the way we make and consume content. Creators now reach a much bigger potential audience, but these relationships are fleeting, one-sided, and often overwhelming.

Kristen Roupenian, reflecting upon her experience after her short story in *The New Yorker*, "Cat Person," went viral, describes it this way:

The word I keep reaching for, even though it seems melodramatic, is *annihilating*. To be faced with all those people thinking and talking about me was like standing alone, at the center of a stadium, while thousands of people screamed at me at the top of their lungs. Not *for* me, *at* me. I guess some people might find this exhilarating. I did not.¹³

Within the world of creators, open source developers are an unusually interesting subset to look at. From an economic perspective, code is similar to other forms of content. Like a book or video, code is just a bunch of information, packaged up for distribution. But its role as a utility is more explicit.

While social media, news, and entertainment all serve critical public roles in our lives, we directly rely on open source code to keep our phones, laptops, cars, banks, and hospitals running smoothly. If a YouTube video goes down, we might lament the loss of valuable knowledge, but if an open source project goes down, it can literally break the internet (as we'll see in Chapter 4).

As a result, examining the behavior of these developers, as compared to other types of creators, prompts a certain set of essential economic questions. It also helps that open source developers work in full public view, making their stories easier to study.

Open source has always served as a vanguard for the rest of our online behavior. In the late 1990s, open source was the poster child for a hopeful vision of widespread public collaboration, then dubbed “peer production.” Because open source software was starting to outpace software sold by companies, economists believed that these developers had achieved the

unthinkable. As the internet floated peacefully in its embryonic state, it really did seem possible that the world might eventually be powered by the efforts of self-organized communities.

But over the last twenty years, open source inexplicably skewed from a collaborative to a solo endeavor. And while more people *use* open source code than ever before, its developers failed to capture the economic value they created: a paradox that makes them just as interesting to reexamine today.

By studying open source's transition from "small internet" to "big internet," we can better understand what's happening to online creators more broadly. We're still trying to reconcile the rise of individual creators with the crumbling of newspapers, book publishers, and talent agencies: the decline of the firm as a principal agent of change. As a case study, open source helps us understand why our online world didn't evolve the way that early scholars predicted, as well as how our economy might reorient itself around individual creators and the platforms upon which they build.

If creators, rather than communities, are poised to become the epicenters of our online social systems, we need a much better understanding of how they work. In a world where 4.5 billion people are now online, what is the role of one? How do these creators shape our tastes, and how do we protect, encourage, and reward that sort of work, long after the glow has faded? How do platforms help or hinder that work?

^{*}This is true for contributions made in 2017.

[†]This data was collected between August 29, 2017, and August 29, 2018.

‡ Even Wikipedia, the world's biggest online encyclopedia, and a canonical example of large-scale collaboration, has Steven Pruitt, a volunteer who's edited one-third of all the English language articles on the site.¹²

PART 1

HOW
PEOPLE
MAKE

GITHUB AS A PLATFORM

01

This book is an attempt to identify, and expand upon, what it means to be online today, told through the story of open source, where individual developers write code consumed by millions. Rather than maximizing participation, their work is defined by the opposite: the need to filter and curate a high volume of interactions. I try to explain why this came about and how platforms shifted the focus from online communities to solo creators.

As I explore this topic, I'll focus primarily on developers who use GitHub.^{*} This is not to diminish the value of open source that is developed outside of GitHub, nor is it meant to imply that open source can only ever be developed on a single platform, or that opportunities to migrate elsewhere should not exist. I'm also mostly writing about the experience of individuals rather than the role of institutions or corporate actors in open source—that's a different, complex topic that deserves its own separate book.

Open source has a long, colorful history of remaining independent of any proprietary software, tool, or platform. Its story predates GitHub by more than twenty years. For the most part, open source's lack of standard tooling was by design: developers like choices, and they enjoy having the flexibility to pick their favorite tools for the job.

GitHub had a meteoric impact on open source. It crashed through the roof of the Church of Free and Open Source Software and landed on the pews, crushing everything beneath.

Although there's no requirement that developers *must* use GitHub to write open source software, GitHub is by far the dominant code-hosting platform today. The fact that most open source developers now use GitHub

represents more than just a shift in personal tastes; it also signifies a shift in developer culture.

The relationship between platforms and their creators is critical to the discussion of how our online world is changing. And because GitHub is the dominant platform for open source, there is no way to understand platform-creator relationships without talking about the service and the developers who use, benefit from, and grapple with it.

THE LIBERATION OF CODE

Before there was a de facto platform for hosting code, most open source code was published as a “tarball” (named for the .tar file, which bundles files together) on some self-hosted, stand-alone website. Developers used mailing lists to communicate and collaborate. And every project managed its contributions a little bit differently. Much like visiting another country, if a developer wanted to contribute, they had to get up to speed on the local customs first.

Developers use version control to track and synchronize their changes, which is particularly important when there are multiple people, many of whom are strangers, working on the same code, in different time zones around the world. So if Alice makes changes to the same line of code as Bob, version control helps developers reconcile those differences and keep them organized without breaking any code.

But Git, the most popular version control system today, was only released in 2005. Before that, developers primarily used *centralized* version control systems, such as Subversion or CVS, if they used version control at all. These systems weren’t designed for decentralized collaboration at scale.

Under centralized version control, developers must commit code back to the same server, but with distributed version control everybody can work on their own copy of the code separately, before synchronizing their changes with everyone else. Git (along with Mercurial, a competing system that launched at nearly the same time) was the first major *distributed* version control system to go mainstream, which made it technically feasible for developers to work independently from one another.

Even after Git's release, however, there still wasn't a standardized developer workflow. To some extent, early free and open source developers enjoyed this cacophony of tools and customs and processes, because it meant that no one tool dominated the space, and nobody could capture full developer mindshare.

Richard Stallman, the MIT hacker who's generally credited with starting the free software movement, was inspired to launch the GNU project, a free software operating system, in 1983, after attempting to customize a Xerox printer in MIT's AI Lab and finding that he could not access or modify its source code.

Stallman wanted to liberate code from proprietary use. The term "free" refers to being able to do what you want with the code, rather than the code being free of charge. (Thus the oft-repeated phrase, attributed to Stallman, "Free as in freedom, not free as in beer," and the occasional use of the Spanish word *libre*, rather than *gratis*, to refer to free software, in order to distinguish between these two meanings.)¹⁵

It's hard to overemphasize the extent to which freedom of code matters to free software developers. Bradley Kuhn, a leader of the nonprofit umbrella organization Software Freedom Conservancy, likens his lifestyle to that of a

vegetarian. Just as a vegetarian doesn't eat meat, to the extent that he's able Bradley doesn't use proprietary software.¹⁶ This means not using websites like Twitter, Medium, YouTube, or GitHub. Code, like livestock, needs liberation from humanity, even at the expense of personal convenience.

To write free software, then, was to be free of the constraints that normally plagued commercial software environments. Free software was counterculture, and it fell right in line with the burgeoning hacker culture of the times.

The term "hacker" was popularized by author Steven Levy, who memorably captured a portrait of the 1980s hacker generation in the book *Hackers: Heroes of the Computer Revolution*. In *Hackers*, Levy profiles a number of well-known programmers of the time, including Bill Gates, Steve Jobs, Steve Wozniak, and Richard Stallman. He suggests that hackers believe in sharing, openness, and decentralization, which he calls the "hacker ethic."¹⁷ According to Levy's portrait, hackers care about improving the world, but don't believe in following the rules to get there.

Hackers are characterized by bravado, showmanship, mischievousness, and a deep mistrust of authority. Hacker culture still lives on today, in the way that beatniks, hippies, and Marxists still exist, but hackers don't capture the software cultural zeitgeist in the same way that they used to. The generational successor to hackers today might be cryptographers and those who dabble in information security: those who flirt with the law, and do so with a wink and a bow.

Although Levy doesn't focus exclusively on free and open source developers in his book, hacker culture in the 1980s and '90s was closely intertwined with the early generation of free and open source software, as

evinced by a trio of leaders: Richard Stallman, Eric S. Raymond, and Linus Torvalds.

Richard Stallman (also known as RMS) was the hacker who kicked off the free software movement at MIT in the 1980s. Eric S. Raymond (also known as ESR), the programmer who helped rebrand free software to “open source” in the 1990s, is widely viewed as early open source’s unofficial anthropologist. And Linus Torvalds is the programmer who created both Linux, the open source kernel that powers many of today’s operating systems, in 1991, and Git, in 2005. The former project became a poster child for the early promise of large-scale collaboration, while the latter inadvertently (perhaps to Torvalds’s chagrin) gave rise to GitHub.¹

If free software is an ideology, Stallman is its devoted, disheveled evangelist. I once heard him described as “the guy who shows up to lectures wearing Hawaiian muumuus and carrying plastic shopping bags.” He’s known for his insistence on distinguishing between “free software” and “open source software,” a bizarre phobia of plants, and showing up uninvited to presentations to grill unwilling lecturers on free software trivia.

Eric S. Raymond is part of a group of programmers who embraced the term “open source,” hoping to distance themselves from free software’s ideological positioning and make the idea seem more commercially friendly. His 1997 book-length essay, “The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary,” makes a compelling case for the benefits of open source software. The essay argues that developers will find more software bugs when the process is highly participatory (like a “bazaar”), compared to when it’s restricted to a smaller group of developers (like a “cathedral”). Raymond and a group of

like-minded developers then formed the Open Source Initiative the following year to evangelize their ideas.[‡]

In addition to writing about open source software, Raymond writes a number of personal essays, which form a brand that is unmistakably ESR. He's notorious for publishing "Sex Tips for Geeks," a collection of posts about picking up women and being good in bed.¹⁹ And as a self-described "gun nut" and libertarian, Raymond writes essays about gun ownership, which are collected on "Eric's Gun Nut Page."²⁰

Rounding out the triumvirate is Linus Torvalds, who's known for a lot of things, but is perhaps most notorious for his brash style of communication, as well as an iron grip over his open source projects. Torvalds's mailing list messages are filled with expletives, sometimes in all caps. The examples are too numerous to include, but here's one: In 2012, he gave a talk at Aalto University in Espoo, Finland. In response to a question about Nvidia's lack of support for Linux (Nvidia is a manufacturer of graphics processing units, or GPUs), he turned to the camera, gave it the middle finger, and growled, "Nvidia, fuck you!"²¹

It's not just Torvalds's communication skills but also his governance style that helped him gain notoriety. In one of his essays, Raymond called this style "benevolent dictator,"²³ which was later adapted by Guido van Rossum, author of the Python programming language, into the better-known phrase "Benevolent Dictator for Life" (BDFL), to describe authors of open source projects who retain control even as the project grows. Although the Linux Foundation reports more than 14,000 contributors to the Linux kernel since 2005,²⁴ Torvalds is still the only person who's allowed to merge those contributions into the main project.²⁵

Although there is no shortage of memorable hacker personalities, the free and early open source ethos is also defined by a startling lack of interest in its people. One of my earliest conversations with a prominent member of the free software movement (who I'll keep anonymous) involved him sputtering angrily at the idea that open source had anything to do with the developers who produce it. He told me that code is "anarchist" and "untouchable," and that it must be able to survive beyond any one person's desires, or their need to pay rent. "It needs to be something that nobody can take away," he said. "It's the system that's important. If one developer goes away, another will step in and maintain it." Freedom of code extends to freedom from the people who make it, too.

Karl Fogel, who coauthored the popular version control system Subversion (a centralized predecessor to Git),[§] echoed this sentiment to me:

When we say: "that's my bike" or "those are my shoes," we mean that we own them. We have the final say in decisions about our bike. But when we say "that's my father" or "my sister," what we mean is we're associated with them. We obviously don't possess them.

In open source, you can only have "my" in the associative sense. There is no possessive "my" in open source.²⁶

It's understandable, then, why many of these developers bristle at the idea of code being tied to a particular platform. Given that open source has its ideological roots in "freeing" code from proprietary control, many developers are concerned about retaining the freedom to publish and collaborate on code anywhere on the web. The ability to move elsewhere is pervasive throughout the design of open source, whether it's *forking*—or

making a copy of—code or being able to *clone*—or download—a project to one’s local computer.

To the chagrin of these developers, GitHub itself is not open source, meaning that, like Stallman with his Xerox printer, users are limited in their ability to modify the platform to suit their own needs. Git, the version control system used to manage code, works independently of GitHub or any other hosting platform, so anyone can copy and host a project’s code elsewhere. But GitHub’s community features—where all the conversations take place—are harder to export. (GitHub’s next-closest competitor, GitLab, which was launched in 2011, heavily advertises the fact that its platform is open source. GitHub is still by far the dominant market player: while it’s hard to find public numbers on GitLab’s adoption, its website claims more than 100,000 organizations use its product,²⁷ whereas GitHub claims more than 2.9 million organizations.²⁸)

Many free software developers refuse to use GitHub altogether. Eric Wong, who maintains the open source web server Unicorn, wrote the following in response to a user’s request to “please move to github”:

No. Never. Github is [*sic*] proprietary communications tool which requires users to accept a terms of service and login. That gives power and influence to a single entity (and a for-profit organization at that). . . . The reason I contribute to Free Software is because I am against any sort of lock-in or proprietary features. It absolutely sickens me to encounter users who seem to be incapable of using git without a proprietary communications tool.²⁹

Both free and early open source advocates were preoccupied with evangelizing the *idea* of open source, whether for ideological or business

reasons. But today's developers hardly even notice "open source" as a concept anymore. They just want to write and publish their code, and GitHub is the platform that makes it easy to do that.

THE TRIUMPH OF CONVENIENCE

GitHub was founded in 2008 by four Ruby developers, who billed it as a “social coding” platform at a time when Web 2.0—Facebook, Twitter, YouTube, Tumblr—was just getting off the ground. By bringing open source developers all to one platform, GitHub helped standardize how they worked.

GitHub wasn’t the first code-hosting platform. It was preceded by SourceForge, founded in 1999. If GitHub is like Facebook, SourceForge was the MySpace of code-hosting platforms: the first significant product of its kind, and, though still alive today, mostly remembered as a blueprint. Like GitHub, SourceForge gave developers a place to host and download code, but it was more like a file-sharing site than a place to collaborate. SourceForge focused on code distribution; GitHub focused on improving the developer workflow. Everybody has an opinion as to why SourceForge didn’t succeed. Many people blame its reliance on advertising, which negatively impacted the user experience, as well as generally poor product development. Others point to SourceForge’s reluctance to support Git as a version control system until it was too late.

As the name implies, GitHub’s founders staked its reputation on Git—only a few years old at the time—as the future of software collaboration. Doubling down on relatively new technology helped GitHub attract a new wave of developers who were just realizing the benefits of Git. (SourceForge, the incumbent code-hosting platform, did not support Git at the time.) GitHub’s friendly user experience helped boost interest in Git, which has a notoriously steep learning curve. It’s hard to say whether Git or

GitHub enabled the other's success, but bundled together they became the dominant toolset for large-scale, distributed software collaboration.³⁰

GitHub incorporated every part of the developer workflow into its product, including issue trackers, developer profiles, and what it calls “pull requests”: a clean way to submit, review, and merge proposed contributions. Compared to hunting down tarballs and mailing lists on the wild World Wide Web, GitHub’s user interface is easy and intuitive. You sign up, choose a username, and use a search bar to find projects. Opening a new issue or pull request is as simple as clicking a big green button.

By 2017, GitHub hosted 25 million public projects, and the service is still growing. GitHub claims that it had more new users in 2018 than in its first six years combined.³¹

By attracting everyone to its platform, GitHub made it easier to discover new projects, as well as every developer’s history and reputation. The site creators added the ability to “star” projects, so you could tell which ones were more or less popular. However flawed the metric may be, stars became—and still are—a mark of success for developers, giving them a way to rank their projects against others’.

GitHub also helped popularize the widespread use of *permissive licensing*, which significantly improved the reach and distribution of open source code. Free software developers frequently champion *copyleft licensing*, like the GNU General Public License (GPL). Copyleft licenses attach themselves, like a virus, to any code that uses them. That means any code containing GPL-licensed code must also be licensed under GPL. So if a company like Facebook uses GPL-licensed code in its mobile app, that app must also be licensed under GPL. As one can imagine, copyleft licenses

aren't commercially friendly, because companies must license their own software on the same terms. So early open source advocates began to emphasize permissive licenses, like the Berkeley Software Distribution (BSD) and the Massachusetts Institute of Technology (MIT) licenses, which allow developers to do pretty much whatever they want with the code, without changing the terms of their own projects.

Today, the MIT license is, by far, the most popular license used by GitHub projects. A 2015 company blog post claimed that 45% of open source projects use it.³² While the MIT license is credited with making the distribution of open source code frictionless, its “set it and forget it” approach has also been criticized for driving the wedge between open source code and its ideological origins. Oddly, the GPL might have been better for open source developers in the long run, since it gives developers more control over how others use their code. But it's difficult to imagine how the GPL would've become the dominant license on a platform like GitHub, because it's incompatible with one of the biggest advantages of platforms: free, unfettered distribution.

GitHub's popularity among modern developers is the classic technology tale of convenience triumphing over personal values. To early free and open source developers, the move toward standard tools and workflows, shepherded by a single company like GitHub (acquired by Microsoft in 2018), represented a backslide against everything they had been fighting for since the 1980s. Code collaboration wasn't supposed to belong to anyone, and especially not to a multibillion-dollar company making proprietary software.

But the GitHub generation of open source developers doesn't see it that way, because they prioritize *convenience* over freedom (unlike free software advocates) or openness (unlike early open source advocates). Members of this generation aren't aware of, nor do they really care about, the distinction between free and open source software. Neither are they fired up about evangelizing the *idea* of open source itself. They just publish their code on GitHub because, as with any other form of online content today, sharing is the default.

Developer Brett Cannon, writing about the decision to move the Python project to GitHub, explains that he and his collaborators chose GitHub over GitLab partly because the former is the platform that developers are most comfortable with today, and partly because it has better tools for automating development. But the third reason he cites is most telling, in terms of differentiating between early and modern open source:

There was no killer feature that GitLab had. Now some would argue that the fact GitLab is open source is its killer feature. But to me, the development process is more important than worrying whether a cloud-based service publishes its source code.³³

GitHub gives developers the tools that they need to do their work. Compared to earlier workflows from the 1980s through the 2000s, GitHub is easy to use, reliable, and capable of handling large-scale interactions. If someone knows how to use GitHub, it's much easier for them to jump into an unfamiliar project and make a contribution. And GitHub users have profiles with photos, bios, and public links to their past activity, all automatically generated, which make their reputation visible across projects.

GNU, the flagship project of free software, doesn't need to be on GitHub to succeed. Given the “code liberation” ideology upon which it was founded, I'd go so far as to say that GNU can only really be popular in defiance of GitHub as a platform. GNU is what it is because it is *not* hosted on GitHub.

By contrast, today's generation of open source developers *needs* GitHub to do their best work. Just as there are Instagram influencers and Twitch streamers, there are GitHub developers. These activities take place away from platforms, too—you can still upload photos or videos of your Hawaii vacation to a self-hosted website—but why would you? For those hoping to reach an audience, platforms and creators have become inseparable.

Platforms are often portrayed as being at odds with creators. *App: The Human Story* is a documentary about Apple App Store developers who struggle against the limitations of their platform.³⁴ Facebook, meanwhile, is frequently accused of “eras[ing] a huge part of publishers' audience” with the “stroke of an algorithm.”³⁵ But for all the problems that platforms might have caused, they've also delivered immeasurable value. Today's open source developers seem to genuinely love GitHub as a place to write, share, and discover code.

Ben Thompson, who writes about business and technology on his blog, *Stratechery*, goes so far as to suggest that delivering this kind of value is, itself, the definition of a platform, as opposed to aggregators. Platforms deliver value to third parties that build on top of them, whereas *aggregators* are pure intermediaries. Thompson references a quote attributed to Bill Gates, who defines a platform as “when the economic value of everybody that uses it exceeds the value of the company that creates it.”³⁶ Based on this definition, Thompson suggests that Facebook is not actually a platform

but an aggregator, because there is “no reason for Facebook, beyond goodwill, to do anything for [media] publishers.”

Like a talent agency, platforms add value to creators by first improving their *distribution*, exposing them to potentially millions of people. The discovery benefit primarily aids creators who are still building an audience. This feedback loop is positive-sum, encouraging more creators to join. So long as more people keep using the platform, there’s no sense that any one creator will ever suck up all the oxygen in the room.

Unlike a talent agency, platforms usually have no contract preventing creators from taking their audience elsewhere. (Some creators are big enough to impact the platform if they do this: top streamer Ninja left Twitch to stream exclusively for Microsoft’s competing platform, Mixer, and Beyoncé made her album *Lemonade* available exclusively on Tidal, a fledgling subscription music service, for the first three years after the album’s release.^{**}) So platforms must make themselves indispensable in order to encourage creators to stay, which usually translates into *reducing costs* for creators. As Brett Cannon explains in his post about migrating Python, GitHub helps his team “automat[e] the development process as much as possible while cutting back on the infrastructure” that Python’s developers have to maintain.³⁷ Python doesn’t necessarily need GitHub for discovery anymore, but as a mature project the programming language especially benefits from GitHub’s cost-reducing infrastructure.

As a result, the platform-creator relationship necessarily resembles a symbiotic, “Russian doll” hybrid production model, where creative talent is widely distributed but nested in the cocoon of a centralized platform. (Anyone can become a creator . . . on Instagram. Or Spotify. Or Medium.)

While creators come from just about anywhere in the world, they need these platforms to grow and survive.

The fact that today's developers don't just use, but actively prefer, GitHub suggests a different set of values from previous generations. The simple act of using GitHub already separates these developers from the most dogmatic free software advocates, who refuse to use the platform whatsoever. The fact that modern developers slap an MIT license onto their projects unthinkingly—if they bother to license their projects at all—separates them from early open source advocates, who built their megaphone on the dogma of “openness,” manifested through permissive licensing.

The GitHub generation of open source developers doesn't feel particularly strongly about these issues. They just want to make things, and sharing is a natural byproduct of those efforts. Jacob Thornton, cocreator of the popular frontend framework Bootstrap, once admitted in a conference talk that he “really had no idea what open source was,” despite working on highly visible open source projects for years.³⁹

Steve Klabnik, a developer known for his work in Rust and Ruby, two popular programming language communities, points to how outdated vocabulary limits our ability to talk about how open source is produced:

Why is it a problem that the concepts of free software and open source are intrinsically tied to licenses? It's that the aims and goals of both of these movements are about *distribution* and therefore *consumption*, but what people care about most today is about the production of software. Software licences regulate *distribution*, but cannot *regulate* production. . . . This is also the main challenge of whatever comes after open source.⁴⁰

By focusing on the developer experience, GitHub made open source much more about people than projects, in what developer Mikeal Rogers calls the “amateurization of open source,” where “pushing code became almost as routine as tweeting”:

I’ve been contributing to open source projects for over 10 years, but what’s different now is that I’m not a “member” of these projects – I’m just a “user,” and contributing a little is a part of being a user.⁴¹

Coding became like tweeting in more ways than one. The advent of package managers provided an easy way for developers to install and manage software libraries associated with their programming language of choice. Package managers first appeared in the 1990s to support the Linux ecosystem, but eventually they became standard tooling for most programming languages. For example, the programming language Ruby has a package manager called RubyGems, while Python has PyPI.

Package managers make it easier to create, publish, and share reusable components for software development, managed through a single registry: imagine a public library, containing thousands of books, which anybody can access with their library card. These managers greatly accelerated what developers were able to do from the command line.

With a single install command, developers could now pull in hundreds of packages—chunks of code written by other developers—and use them in their own code. As a result, the idea of what might constitute an “open source project” became smaller, too, not unlike the shift from blog posts to tweets.

Developer Russ Cox, describing this shift, explains,

Before dependency managers, publishing an eight-line code library would have been unthinkable: too much overhead for too little benefit. But NPM [JavaScript’s package manager] has driven the overhead approximately to zero, with the result that nearly-trivial functionality can be packaged and reused.^{[42](#)}

It’s unsurprising, then, that these emerging behaviors would become among the defining characteristics of JavaScript, a language ecosystem that grew up on GitHub.

FROM HACKERS TO HUSKIES

JavaScript has been around since 1995, but recent technological developments gave it fresh contemporary meaning. It was originally written by Brendan Eich for Netscape Navigator. Because JavaScript is the only language that modern browsers have a built-in engine to support, it has stayed relevant to developers ever since.

Though JavaScript is more than two decades old, for much of its history it was primarily used as a scripting language for webpages. Along with HTML, a markup language, and CSS, a style sheet language, it formed the holy trinity of frontend development. HTML provided the scaffolding for webpages, CSS styled those elements, and JavaScript made them dynamic. If a webpage is a building, HTML forms the bones, CSS the paint and drywall, and JavaScript the electricity and plumbing.

JavaScript was easy to write and tinker with, because developers could do it right from their browser. Every major browser has a “View Source” option, which makes it easy to inspect the HTML, CSS, and JavaScript that make

up any webpage. For many software developers, JavaScript provided them with an accessible first entry point into programming.

In the mid-2000s, developers started finding creative ways to use JavaScript on the backend of their applications (meaning, the underlying logic that is invisible to users). In 2009, Ryan Dahl released Node.js, a platform that made it easier to run JavaScript on both the client and server side. Now, instead of having to learn one language for the backend and another for the frontend of an application, developers could learn one language—JavaScript—and use it everywhere.

Today, GitHub projects are developed in a wide variety of programming languages, including Java, Ruby, and PHP, but JavaScript dominates more than any other language. On GitHub, JavaScript is more than twice as popular as Python, the second-place contender.⁴³ It has rapidly grown to become the most common programming language among developers, according to the site Stack Overflow.⁴⁴

JavaScript's universal appeal makes it both extremely accessible and extremely powerful. As such, from a cultural perspective, it has made strange bedfellows out of frontend and backend developers. Contemporary JavaScript grew up in the post-Web 2.0 era. It's friendly and polished, but also politically charged. It attracts developers who like to explain things, and explain them with emoji and brightly colored logos.

JavaScript developers embrace the “amateurization of open source,” despite garnering occasional eye rolls from their predecessors. Npm, JavaScript’s package manager, encourages modular design by making it easier to both publish and depend on many packages at once. And because JavaScript must maintain compatibility across browsers, its official core library is

smaller than that of most other languages. So developers are forced to customize it for themselves, writing their own packages to fill in the gaps. As a result, each project tends to be smaller and more disposable, like LEGO blocks that fit together instead of a castle carved from stone.

Free software hackers define themselves by independence from platforms, but JavaScript might not have gotten as popular as it did without the advantages provided by a platform like GitHub, not to mention modern communication channels like Twitter and Slack. Unlike those who stress the freedom of code over all else, the JavaScript community exudes the opposite appeal: when code is small, it's the people who stand out.

JavaScript's most recognizable developers are known for the talks they give, the videos they record, the tweets and blog posts they write. They command large followings and attract eager audiences in a way that, say, PHP developers just don't. Kent C. Dodds, for example, spent so much time making content about React that he left his job as a JavaScript engineer at PayPal to become a full-time educator.⁴⁵

JavaScript developers aren't always known for a specific project anymore, either, the way Stallman is known for GNU, or Torvalds is known for Git and Linux. Rather than being associated with just one project, or a few, prominent JavaScript developers often create hundreds of small yet widely used projects. They're known for who they are, rather than which projects they're involved in.

When Antoni Kepinski, a teenage developer who made an open source pizza-tracking app, was asked by an interviewer how he got into programming, he responded,

Actually, I started programming in C#, but this wasn't really my thing, and I stumbled upon a GitHub profile from a user called Sindre Sorhus. When I first saw JavaScript code, I knew it was something I want to learn in the future. That's how it started.⁴⁶

Later in the interview, he joked about using one of Sorhus's projects over that of another JavaScript developer, Feross Aboukhadijeh:

ANTONI KEPINSKI: Sorry, I'm not using your linter [a tool for catching typographic errors in code].

FEROSS ABOUKHADIJEH: Yeah, I noticed that. You're using Sindre Sorhus's, because I think you like him better That's fine. [laughter]

ANTONI KEPINSKI: No, no . . .

MIKEAL ROGERS: That is what it's about. You follow your favorite developer and you're like, "I want their style guide." That is how it goes.

The cult of personality thrives in JavaScript, even if its most prominent developers are reluctant to acknowledge it, perhaps because this attitude clashes with the professed ideal of open source as "community-built." Compared to early renowned hackers like Torvalds, Raymond, and Stallman, many of today's JavaScript developers are unusually humble.

A few examples: Dan Abramov, a maintainer of React, a popular frontend framework, wrote a blog post that lists all the programming topics that "people often wrongly assume that I know."⁴⁷ Henry Zhu, maintainer of Babel, a library that helps translate between versions of JavaScript, says open source helps him "get tested in my patience, my humility, and learn to focus and communicate better."⁴⁸ Sarah Drasner, a maintainer of Vue,

another popular frontend framework, cowrote an “open source etiquette guide” with Kent C. Dodds, in which they emphasize that contributors should “be polite, respectful, and kind.”⁴⁹ Sindre Sorhus’s Patreon page, used to solicit sponsorships for his work, shows him cuddling three happy huskies;⁵⁰ Feross Aboukhadijeh’s Patreon page shows him cozying up with a soft, fluffy kitten.⁵¹ Each of these developers commands a large audience of people who follow them *personally*; they have the attention of thousands of developers, and they choose to use that influence to demonstrate kindness and respect.



Linus Torvalds and Sindre Sorhus. Images used with permission of Aalto University and Sindre Sorhus.

Is every JavaScript developer a saint? Of course not. Like any other open source community, JavaScript gets its fair share of flame wars, spats, and personal takedowns on Twitter and GitHub. But the number of prominent developers who *don’t* engage in these things, and who even seem to shrink away from their celebrity status, is just as remarkable.

In 2019, drama erupted between two subsets of the JavaScript community, Vue and React. Rather than gleefully stoking the flames, as we might imagine prior open source leaders would have done, Dan Abramov temporarily deactivated his Twitter account, later explaining, “I’ve been getting progressively more anxious over the weekend. I’m of no use to this community in that state. I feel much better now, and I’m here to listen.”⁵²

The locus of open source culture has shifted from celebrating heavy-handed authoritarianism to seeking out thoughtfulness and stewardship. Even Linus Torvalds issued a mea culpa in 2018, stepping away from the Linux project for a month in order to reflect on years of “people in our community confront[ing] me about my lifetime of not understanding emotions,” and calling it a “‘look in the mirror’ moment.”⁵³ Eric S. Raymond was banned from the OSI’s mailing lists for combative language.⁵⁴ And Richard Stallman resigned from his positions at MIT and the Free Software Foundation after making controversial comments on MIT’s Computer Science and Artificial Intelligence Laboratory (CSAIL) mailing list.⁵⁵ Many people, on both sides of the issue, noted that Stallman’s comments—in which he dissected the definition of “rape” with the same pedantry that he dissects the term “free software”—weren’t unusually out of character from his past behavior. It’s the times that are changing.

But this proclivity toward kindness also creates tension with the distribution power of platforms. Because of its trendiness, JavaScript attracts inexperienced developers who are learning how to code for the first time, and who also aren’t familiar with how to interact with an open source project.

As a result, the interactions on GitHub between JavaScript maintainers and users can be difficult to manage. Because GitHub is so easy to use, the hurdle to opening an issue or asking questions is low, and users ask maintainers for help with *everything*: how to resolve merge conflicts, how to write a test. These skills aren't specific to any one open source project; they're general "how do I open source" questions.

Sindre Sorhus, who maintains hundreds of JavaScript projects, once tweeted, "After having reviewed 10k+ pull requests on GitHub . . . ~80% of contributors doesn't [sic] know how to resolve a merge conflict."⁵⁶ A few years later, he followed up with more observations:

- Almost no one writes a good pull request title
- More than half don't know about the 'Fixes #112' syntax
- ~30% don't run tests locally before submitting a PR
- ~40% don't include docs/tests⁵⁷

I've also noticed that the general PR quality has gone considerably down in the past few years. I guess it's a result of GitHub's increased popularity. ``_(ツ)_/``⁵⁸

It's not the fault of new developers. They don't know how open source works, and they've been told that they're doing the right thing by asking questions. "Don't let this be a deterrent for you to contribute though," Sorhus himself added.⁵⁹ "Just keep in mind that my time is finite and if I have to go back and forth on your PR for stuff you could have caught yourself with a second look, you take time away from other PRs."⁶⁰

Open source maintainers have become the de facto teachers for developers who are learning how to contribute. In the past, this made sense when new

developers were trying to join a project's community. Today, rehashing the basics to a revolving door of strangers can be fatiguing: death by a thousand paper cuts. Developer Nolan Lawson describes his experience as "a perverse effect where, the more successful you are, the more you get 'punished' with GitHub notifications."⁶¹

THE GITHUB GENERATION

Open source was founded on the idea that developers shouldn't be beholden to a particular platform. More than any other type of creator, open source developers seem like they should've been *most* impervious to platform effects. After all, there were two entire generations of free and open source developers that preceded GitHub, using tools that worked just fine without GitHub, and there were technically savvy users with a high tolerance for inconvenient, but values-aligned, solutions.

And yet, GitHub continues to dominate. Even the Apache Software Foundation, an umbrella organization formed in 1999, which was once perceived as reluctant to adopt modern open source tooling,⁶² announced in 2019 that it was migrating its Git-based projects to GitHub.⁶³

One can lament the death of truly “open” software development, as some holdouts still do, and criticize GitHub for “ruining” the trajectory of open source. But the fact that GitHub became the preferred platform for most developers, overcoming initial opposition that bordered on religious fervor, suggests that maybe platforms do a lot more for creators than we realize. Instead of treating platforms as adversaries, we might think of platforms as powerful allies for creators, and try to understand the strange, symbiotic relationship they’ve formed with one another. Inevitably, such close ties breed strong emotions and conflicts, but maybe that’s a sign that they’ve become indispensable.

The advantages of GitHub—that it’s easy to use, and easy to share and discover others’ code—are also the source of today’s challenges in open

source. These challenges are especially magnified in JavaScript, which attracts developers who are new to software development.

Consider the fact that JavaScript projects are deliberately small and modular, instead of big community projects with longtime members who are able to onboard newcomers. Then consider the reluctance of JavaScript leaders to appear rude or unwelcoming, having turned their backs on the bird-flipping, gun-touting specters of Stallman-Torvalds-Raymond's hacker diaspora.

Taken together, it's a perfect storm of drive-by users—most well intentioned, others not so much—flooding the channels of single maintainers, using a platform that's designed to *encourage* this behavior, and shielded by social norms that explicitly discourage pushing back on inbound requests.

Like every other social platform today, GitHub was designed for a use case that's breaking down at scale. Every platform must figure out how to adapt to an emerging set of social behaviors that is still not well understood.

^{*} My scope is also generally limited to the United States, Europe, and Australia. Open source development is growing outside of these areas, especially in China and India, but because their behavior differs in certain ways, I've limited my focus to geographic regions that I can speak to more confidently.¹⁴

^t In the past, Torvalds has refused to accept pull requests to the Linux kernel from GitHub, calling it “fine for *hosting*, but the pull requests and the online commit editing, are just pure garbage.”¹⁸

[‡] While the definitions of “free” and “open source” software are, even according to Stallman himself, essentially the same, the two concepts are culturally distinct. Some people, particularly free software developers, don’t like to mix these terms. Free software advocates are united by ideology, “a movement for freedom and justice.” They believe that code should be liberated from proprietary control. Early open source advocates, on the other hand, focused on pragmatic goals, such as standard licensing to make it easier for code to be freely distributed and used by anyone, including commercial entities.²²

[§] Fogel is a thoughtful, and in my view underrated, voice of early open source culture. He wrote the only notable book to date on the production of open source: *Producing Open Source Software: How to Run a Successful Free Software Project*, first published in 2005. It’s available in its entirety online at <https://producingoss.com/>.

[¶] According to Stack Overflow’s 2018 developer survey, 87% of respondents use Git.

^{**} From Beyoncé, rapping on the song “Nice”: “Patiently waiting for my demise / ‘Cause my success can’t be quantified / If I gave two fucks, two fucks about streaming numbers / Woulda put Lemonade up on Spotify.”³⁸

THE STRUCTURE OF AN OPEN SOURCE PROJECT

02

In conservation biology, the term *charismatic megafauna* refers to the idea that polar bears sell environmental causes better than mollusks or insects. The cuter, the better. In the world of creators, filled with a veritable forest of charismatic megafauna, like YouTube comedians and Instagram models, I picked the fluted kidneyshell.^{*}

Open source is complicated because it contains a messy mix of both technical and social norms, most of which play out in public. It is documented *extensively* (nearly every decision is written down somewhere) but not *clearly* (you have to dig through years of mailing list archives to find what you need). Its treasures are hidden amidst a tangle of brambles and thorns.

Social norms are passed down through trial and error, which means that getting something wrong runs the risk of embarrassment and mockery in front of one's peers. Developers don't contribute to open source for lack of technical ability, but rather due to fear of committing a faux pas.

Infrastructure developer Julia Evans, who's well known for her code and words, writes of her experience contributing to open source:

I sometimes feel kind of intimidated by open source. . . . In open source, I need to send code reviews to total strangers. At work, I generally send code reviews to the same 10 people or so, most of whom I've worked with for a year or more, and who often already know exactly what I'm working on.⁶⁴

To make things even more challenging, these technical and social norms differ between language ecosystems, types of projects, and even individual developers. (One developer described this to me as “one size barely fits

most.”) Knowing how to participate in open source as a Python developer does not mean you can saunter into the world of Haskell. A C++ developer might feel out of place among a group of Electron developers.

The tribalistic nature of open source is frequently overlooked and misunderstood. As we’ve seen from the evolution of “hackers” to the more neutral “programmers” or “engineers,” to the polished-sounding “developers” today, there is no “open source community,” really, anymore than there is an “urban community.” Certainly, one city dweller can sympathize with another in many ways, especially when comparing themselves to their suburban or rural counterparts. But knowing the streets of San Francisco says nothing about how well you’ll fare in Hong Kong. “Urban” is an adjective, not an end point.

To carry the analogy further: There are cities, and then there are cities.[†] At roughly 200,000 residents, the city of Reykjavik is the largest in Iceland.⁶⁶ But it would barely register as a city in China, where the largest city, Shanghai, counts over 24 million residents.⁶⁷ Likewise, there are open source “projects” like chalk—a tool for styling one’s code with colors and text formatting—which contain few lines of code and perform a small, but useful, function. There are also open source projects like OpenStack—a software platform for cloud computing—with millions of lines of code, broken into large subprojects that each dwarf the codebase of entire stand-alone modules.

The term “open source” refers only to how code is distributed and consumed. It says nothing about how code is produced. “Open source” projects have nothing more in common with one another than “companies” do. All companies, by definition, produce something of value that is

exchanged for money, but we don't assume that every company has the same business model.[‡]

This book doesn't assume you are a developer, but it does assume that you're not afraid to learn. For those who are less familiar with open source software, I'll give a brief overview of how these projects are structured, which will help decode the interpersonal dynamics within.

HOW CONTRIBUTIONS ARE MADE

Open source software is frequently characterized as participatory, which implies that anyone can modify its code. While this is theoretically true, in practice open source is not blindly open to every person who wants to change it. (By contrast, wikis, for example, are usually editable by the general public, with no additional permissions needed.)

Anyone can propose a change to an open source project in the form of a “patch”—or to use GitHub's nomenclature, a *pull request*—but these changes undergo reviews and are subject to approval from prior trusted contributors. It's the equivalent of having comment permissions on a shared document: anyone can suggest a change, but not everyone can actually approve it.

Some developers have permission to merge changes into the trunk (or *master*), which is the baseline version of the project. Having these permissions is often referred to as *commit access*, which is like being able to edit a shared document.

Commit access is a technical permission, but there are also social considerations. Even those who have commit access cannot wield their

power unilaterally. Before they merge a change in, they must also consider how it will be received by other contributors and users.

Bigger projects often use a formal “request for comments” (RFC) process to allow communities to discuss these changes before they are merged. In Python, for example, these requests are called Python Enhancement Proposals (PEPs),⁶⁸ while in Go, another programming language, a formal proposal is called a “design document.”⁶⁹ On smaller projects, the RFC process might just look like an informal discussion thread on an open pull request.

Conversely, there are also maintainers who are socially viewed as leaders and are influential to the project, but also don’t have commit access. In some projects, nobody gets commit access besides the author, no matter how big the project gets. Alex Miller, for example, is a longtime maintainer of the programming language Clojure, but he doesn’t merge patches. Instead, he triages and uplevels patches from the community, which are then reviewed and merged by a few maintainers with commit access, primarily Rich Hickey—Clojure’s author and lead developer—and Stuart Halloway, another co-maintainer.

Brett Cannon remembers his experience gaining commit access to Python:

I had been regularly submitting patches for months when I said I would happily write some documentation as long as someone committed the patch. Guido [who authored Python] replied, “Do you have a SF userid yet? Then we can give you commit privs!” I gained my commit privileges like people still do today on the Python project: I consistently contributed good patches and eventually a core developer noticed and asked if I wanted to join the team.⁷⁰

The process by which a developer gains commit access varies widely between projects and is subject to preexisting social norms. Some projects adopt the mindset of needing to demonstrate trust, while others prefer to operate on the basis of trustlessness.

For example, Debian, an operating system based on Linux, requires that developers follow an extensive onboarding process in which they are expected to read a manual, find a mentor, and meet a maintainer in person who can vouch for their identity.⁷¹ On the other hand, it's common among JavaScript developers to give away commit access more freely. The idea is to distribute the burden of maintenance by making it easy for others to contribute, and it's assumed that strangers are trustworthy until proven otherwise.

These differences in social norms are often closely intertwined with technical design. Clojure's core developers highly prioritize stable code, which means they're more reluctant to accept changes.⁷² And Debian has a monolithic, tightly coupled codebase, where green-lighting the wrong maintainer could, indeed, have dire consequences. But JavaScript, including Node.js, is designed to be modular, where each maintainer has a limited ability to affect other components of the ecosystem, so JavaScript developers are more likely to prioritize moving fast and accepting contributions.

In cryptocurrency, these philosophies play out as visible differences in how the Bitcoin and Ethereum projects are managed. Bitcoin's community, like Clojure's, prioritizes stability and security, preferring to move slowly and with care, even if it means including fewer features and contributors. Ethereum is more like Node.js: it's a platform for others to develop on,

flinging itself far and wide. It resembles a sprawling city like Los Angeles, comprised of many neighborhoods and subcultures. Despite impassioned rants to the contrary (if there's one thing I've learned, it's that developers have *opinions*), there's no one right way of doing things, just different communities that each have their own cultural norms.

The process of getting a change approved depends, among other things, on the *complexity* of the change (and the complexity of the project), as well as on one's *reputation* among those with the ability to approve the change.

It helps if developers propose a change that aligns with the project's goals. For example, they may have tackled a wish list feature that was previously discussed in public, or fixed a known bug on the project. These changes are often easier to get approved than radically new ideas are, because the need is more obvious.

A screenshot of a GitHub pull request comment. The comment is from user 'dkubb' and is dated 'Apr 18, 2016'. The comment text reads: '@backus this is awesome! Feel free to ping me for reviews about this anytime; this is one of the features I've wanted most in mutant and I'd love to help our however I can.' Below this, another comment from 'dkubb' says: 'I would say something like this should be merged early once you've got one or two basic mutations working reliable and @mbj has approved it. Your work provides an outline, and we can gradually fill in various other mutations over time.' There are 'Collaborator' and emoji buttons next to the comments.

Response from a maintainer of Mutant, a testing tool for Ruby developers, to a new contributor.⁷³

A developer's reputation can strongly influence whether a pull request gets merged. Reputation is not always limited to the specific project but also the wider ecosystem. (Of course, this also works in reverse: a poor reputation

elsewhere can carry over to a project and make it harder to get a contribution merged, even if the person has never contributed to that particular project before.)

Lorenzo Sciandra, for example, became a maintainer on React Native, a framework for building mobile apps with React, by first contributing to React Navigation, a related library.⁷⁴ At the time, he didn't feel "good enough" to contribute to React Native, but when React Navigation's maintainer left the project, the remaining developers were actively recruiting more contributors.

Sciandra didn't feel confident enough to write code, so he started closing issues instead, tackling over 900 issues in four months. His activity caught the attention of Héctor, a React Native maintainer working at Facebook, who asked Sciandra if he wanted to contribute to the project, based on his React Navigation experience.

There is no guarantee that someone with commit access will review, or even acknowledge, a developer's pull request, which can lead to governance disputes. A company, for example, might release open source code but rely primarily on its own employees to maintain it. Although the project is open source in the sense that anybody can use, inspect, fork, and modify the code, it might be difficult to make substantial contributions as a non-employee. Anybody can submit a change, but that doesn't mean it will be approved.

While governance disputes might seem more visible when corporate actors are involved, these issues occur on single-maintainer projects, too.

Setuptools is a widely used library for Python packages that was largely maintained by its author. After many users complained that it was difficult

to get their contributions merged into Setuptools, a group of developers forked the project into their own version, called Distribute, which was eventually merged back into Setuptools several years later.

If a developer is otherwise unknown in the project, they must try to lobby for a maintainer’s attention. The polite thing to do is to contain one’s interactions to the specific pull request. It’s acceptable to @-mention (i.e., notify) the relevant person on the thread to ask for a review, so long as the request isn’t spammy—much like the practice of following up on an unanswered cold email after a week rather than a day. Reaching out through personal channels, such as private email, is generally considered rude, in the same way that tweeting at someone to respond to your email is considered rude, unless otherwise specified.

Amjith Ramanujam, for example, lists his email and Twitter handle in the README of his project, pgcli, a command-line interface for PostgreSQL, an open source database system, adding that contributors should “please feel free to reach out to me if you need help.”⁷⁵ By contrast, WP-CLI, a command-line interface for the popular content management system WordPress, requests the exact opposite in its README:

Please do not ask support questions on Twitter. Twitter isn’t an acceptable venue for support because: 1) it’s hard to hold conversations in under 280 characters, and 2) Twitter isn’t a place where someone with your same question can search for an answer in a prior conversation.⁷⁶

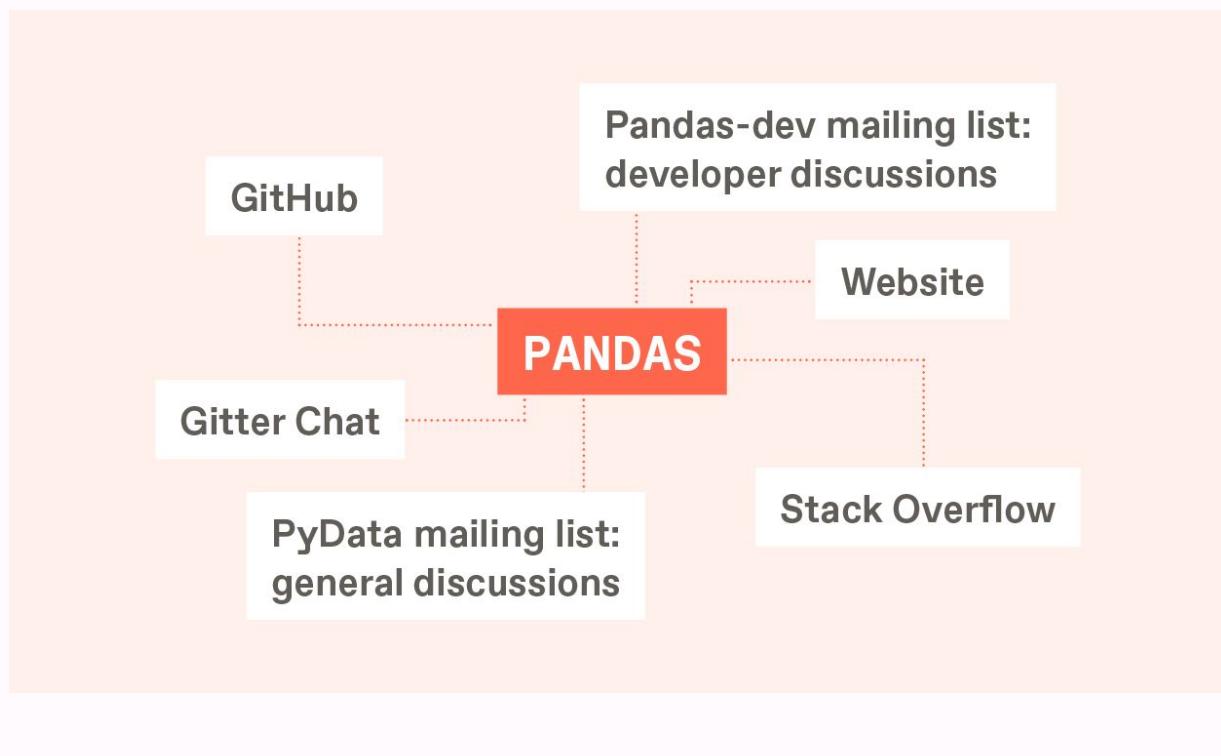
Depending on the size, complexity, and popularity of the project, as well as one’s proposed change and how much anyone is paying attention, pull requests can take anywhere from minutes to months to get a response.

Sometimes, they’re merged within an hour. Other times, they are never reviewed at all.

WHERE INTERACTIONS TAKE PLACE

There’s a reason why open source projects are called “projects,” rather than just code. While code is the final output of a project, the term “project” refers to the entire bundle of community, code, and communication and developer tools that support its underlying production.

The terms “project” and “repository” are sometimes used interchangeably, and repositories often serve as the main namespace of a project. But *repository* refers specifically to the file directory that contains code, whereas *project* implies the full set of tools and communication channels that support the code (e.g., mailing lists, chat, documentation, and Q&A sites).



Pandas, a Python library for data analysis, extends across multiple communication channels.

Repositories are a single unit of measure, but vary greatly in terms of granularity. Is-sorted, a module that checks if an array is sorted and contains less than thirty lines of code, counts as one repository.⁷⁷ So does CPython, a repository that contains the entire codebase for the Python programming language.⁷⁸

As with anything else, the size and structure of a repository are influenced by technical decisions and personal preferences. Some projects like to break up their code into many repositories, each of which contains a smaller amount of code. These repositories can all live under a single GitHub organization. Other projects adopt the *monorepo* philosophy, managing large amounts of code under a single repository.

The Python programming language <https://www.python.org/>

106,208 commits	12 branches	0 packages	413 releases	1,124 contributors	View license
Branch: master ▾	New pull request	Create new file	Upload files	Find file	Clone or download ▾
<p>pablogsal and tim-one bpo-17005: Add a class to perform topological sorting... ... Latest commit 99e6c26 1 hour ago</p> <p>.azure-pipelines Fix Windows release builds (GH017550) 2 months ago</p> <p>.github Run doctests in GitHub actions Docs targer (GH018041) 6 days ago</p> <p>Doc bop-17005: Add a class to perform topological sorting to ... 1 hour ago</p> <p>Grammar bpo-35814: Allow unpacking in r.h.s of annotated assignment ... 8 months ago</p>					

GitHub repository for CPython.⁷⁹

At minimum, open source projects hosted on GitHub can be broken into three parts: *code* (the final output of a project), an *issue tracker* (a way to discuss changes), and *pull requests* (a way to make changes).

Code is typically managed using a version control system, of which Git is the most popular. This system is bundled directly with the code, using a .git file directory, which means that changes will be tracked irrespective of where those files are actually hosted, whether on GitHub or elsewhere.

Issues and pull requests, on the other hand, live on GitHub. Although the concepts of an issue (also called a *ticket*) and a pull request (also called a *patch*) are much older than GitHub, issues and pull requests are GitHub's branding of these features, and therefore aren't quite so easy to migrate between platforms. Issue trackers are used for conversations, like discussing new features or reporting bugs. Pull requests are proposed changes, which, if approved and merged, will modify the actual codebase.

Issues tend to fall into three categories: *bug reports*, *feature requests*, and *questions*. From a maintainer's perspective, *bug reports* are highest priority, because they mean that something isn't currently working as intended. *Feature requests* are things that users would like to have. *Questions* are essentially support requests, e.g., "How do I do X?" Some maintainers do not use the issue tracker for questions, preferring instead to direct users off GitHub to other support channels.

Some open source projects, particularly bigger ones, might use GitHub to host and manage their code but use different tools to manage other aspects of the project. For example, Django hosts its code on GitHub but uses a different product, called Trac, for its issue tracker. React also hosts its code

on GitHub but keeps its documentation, tutorials, and community on a separate website.⁸⁰

Projects often utilize both *synchronous* communication channels (like IRC, Slack, or Discord) and *asynchronous* ones (like mailing lists, Reddit, or GitHub issues) in order to talk to users and other contributors. These channels are used for all sorts of conversations, including:

- Asking and answering questions among both users and contributors (“How do I use the project to do X?” as well as “How do I make Y contribution?”)
- Coordinating work among maintainers
- Providing a community space
- Holding office hours
- Educating users (e.g., teaching or demonstrating something)

Stack Overflow, a Q&A website for developers, became a significant complementary tool to GitHub (though its usefulness varies depending on the programming language or framework) because it is where users often ask questions and receive answers; the site has its own social dynamics and reward system. Users get points for answering questions, which builds their public reputation. A top answerer on Stack Overflow might never interact with the project’s core developers on GitHub, despite providing significant support to their users; the two platforms are separate ecosystems associated with the same project. (The topic of user-to-user support systems will be covered in more depth later on.)

HOW PROJECTS CHANGE OVER TIME

The relationship between a project's maintainers and their community changes depending on the maturity of the project. At a high level, open source projects tend to move from *closed* —> *open* —> either *closed* or *distributed* development (depending on their size).

CREATION

In the earliest stages of a project, there are probably one or a few developers who are writing code in a fairly closed state of development. While some open source developers write code in public from the very beginning, many prefer to do their initial creative work in private, so they can properly articulate their ideas before opening the project up for feedback. Even if developers do publish their code early on, they may not advertise it widely until they have something ready for release.

Ryan Dahl, who created Node.js, the popular JavaScript platform, recalls his early experience working alone for six months, eventually presenting his first demonstration at the conference JSConf EU:

I quit my work and I worked on Node for six months, basically straight. I was completely convinced that this would be a thing. I wrote very nicely to the JSConf people and begged them to give me a slot where I could present it at JSConf EU. . . . I was extremely scared, presenting this thing that I had been working on for six months.^{[81](#)}

EVANGELISM

Once the project is released, the author or authors are usually eager to get feedback, bug reports, issues, and pull requests. They'll often promote the project just like a founder would promote a startup: by sharing it on relevant channels online, giving talks at conferences and meetups, and encouraging others to write and talk about it.

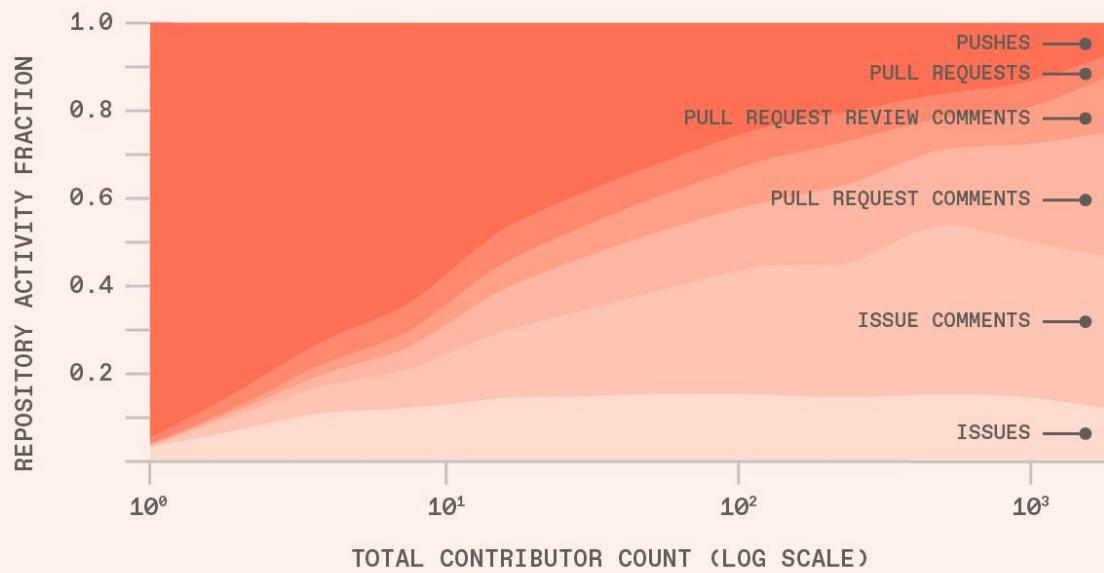
Nathan Marz, who wrote the data-computation system Apache Storm, timed its release with his talk at a software conference called Strange Loop, open-

sourcing the project live onstage. The announcement got a lot of attention, and he set up a mailing list for developers to give feedback, where he “spen[t] one to two hours a day answering questions.”⁸² Over the next year, he continued giving talks about Storm at conferences, meetups, and companies, noting that “all this speaking got Storm more and more exposure.”

At this stage, the goal is *distribution*—getting new code in the hands of other developers—so the project transitions to a more open state of development. As when publishing a blog post or a video online, open source developers want engagement with their material, so they’ll encourage contributions while still retaining control over their vision of the project. If other developers are filing issues, starring the project, and asking questions, it’s a sign that people find it interesting.

Projects that do attract attention from other developers may stabilize here, with a solid, if not rapidly growing, group of users who continue to download and use the project. For a smaller subset of projects, however, user and contributor interactions will continue to rise, pushing them into the *growth* stage.

Repository activity as a function of total contributor count



From “The Shape of Open Source,” by Arfon Smith. As contributor count increases, the type of activity on a repository changes, particularly in the form of comments.⁸³

GROWTH

As a project becomes more widely used, more developers will interact with it. One heuristic for when this transition occurs is when maintainers start doing more non-code than code work on the project, such as triaging issues and reviewing others’ pull requests.

“Growth” means that more developers might be *using* the project, but that doesn’t necessarily mean that more developers are making meaningful contributions, or that there are more maintainers meeting this level of demand (as one might expect a successful company to grow employee head count). At this stage, community interactions get noisier—comments,

feature requests, pull requests from strangers—which makes it harder for maintainers to be responsive to everyone.

Felix Krause, who wrote fastlane, a tool for mobile developers, explains,

The bigger your project becomes, the harder it is to keep the innovation you had in the beginning of your project. Suddenly you have to consider hundreds of different use-cases

Once you pass a few thousand active users, you'll notice that helping your users takes more time than actually working on your project. People submit all kinds of issues, most of them aren't actually issues, but feature requests or questions.⁸⁴

How a maintainer meets this demand for attention depends on their type of contributor base. (These two paths will be discussed extensively in Chapter 5.) If they don't have many contributors, they'll start to filter out the noise, pulling back to a more closed, focused state of development. In a *closed* state, maintainers are more selective about reviewing external contributions, so they can focus on their work.

If a project's contributor base is growing rapidly, and there is enough work to hand off, maintainers may start to distribute work more widely. In a *distributed* state, maintainers actively recruit more contributors to pitch in, with the goal of retaining them in the project. In this case, maintainers are making an additional investment of time into growing the number of developers who are available to respond to user and contributor demand.

CLASSIFYING PROJECT TYPES

As a project grows, it acquires more users and contributors. Starting from the assumption that open source is participatory, we can imagine a “perfect” user-contributor relationship as 1:1, where each additional user is also a contributor.

In practice, a project’s user and contributor bases grow at different rates, sometimes independently of one another. Some projects seem to have a lot of users but not many contributors. Others have an active contributor community, but they don’t seem to be very widely used. Which factors contribute to these differences?

A project’s contributor growth is a function of its *technical scope*, *support required*, *ease of participation*, and *user adoption*.

Technical scope refers to the size and complexity of a project’s codebase: in other words, how much more there is to do. A project that feels feature-complete won’t attract as many contributors as one that’s extensible and customizable. It could be widely *used*, but that doesn’t mean there’s a lot to *contribute* to.

For example, React is a JavaScript library for frontend development that can be thought of as a platform, because it forms the base upon which many other applications are built. Webpack, a tool for bundling JavaScript files, is one component of that “platform”: it’s frequently used by React developers, but its purpose is much more tightly scoped. It’s easier to imagine how a developer might regularly contribute to the React ecosystem than to webpack specifically. Webpack is voluntarily maintained by four developers (with just one core developer, Tobias Koppers),⁸⁵ whereas React is maintained by a larger team, many of whom are employed by Facebook.

Support required refers not just to writing code for the project but also to supporting tasks, like responding to open issues or reviewing others' pull requests. The former is considered to be "fun" work that potential contributors clamor to take on, but the latter is necessary work that tends to fall exclusively on maintainers. It's much easier to entice a new contributor to write an interesting new feature than it is to ask them to triage issues.

In addition, tasks like issue triage require more familiarity with the project than a casual contributor might possess. Someone who's worked on a project for years will be able to recognize duplicate issues or commonly asked questions more quickly than a new contributor would.

A project could have a small codebase but require a lot of user support, and vice versa. For example, `Youtube-dl`, a program that allows users to download videos from YouTube and other video sites, is a fairly "small" project in terms of technical scope, but it has one of the highest support volumes on GitHub.⁸⁶ `Font Awesome`, a popular icon toolkit, is another project that is tightly scoped from a code perspective, but it receives many new icon requests and contributions from its users.⁸⁷

Ease of participation refers to how easy it is for someone to contribute to the project. Whether a project is easy to contribute to is partly a function of its technical scope, but there are other factors involved: the quality of its documentation, how responsive the maintainers are, the perceived hostility (or lack thereof) in social interactions, and the tools or preexisting skills required to make a contribution. Most important is the question of whether the project is on GitHub or not.

When it comes to attracting new contributors, there is a great divide between GitHub and non-GitHub projects. One infrastructure developer,

who worked for years on projects with different tooling, told me that he's now so used to GitHub that if he finds a bug on a project that uses a different issue tracker, he won't even bother filing an issue anymore—it's too much work.

Babel experimented with moving its issues to Phabricator in 2015, after its author, Sebastian McKenzie, complained on Twitter that "GitHub is such a poor tool for managing large open source. I constantly feel overwhelmed, the UI [user interface] does not scale well at all."⁸⁸ Less than a year later, the library's developers moved their issue tracker back to GitHub. While Phabricator may have been a more powerful tool, people weren't as familiar with it. Rather than adapt to a new issue tracker, Babel's users would instead report their issues on Twitter, in unrelated repositories, and in commits. Other users felt intimidated by going to a separate website to report issues. At the time, Babel's maintainers noted, "We have very few consistent contributors and like most projects a few maintainers. If we're [sic] going to grow our contributors, we should at least lower the barriers and try to work within Github."⁸⁹

User adoption refers to a project's reach: What is the total addressable market of contributors? How many people *could* potentially contribute to this project? Since most contributors start out as users of an open source project, user adoption is one heuristic for estimating a project's total potential contributor base.

For example, although there is a strong, active community of developers who love to write in the programming language Go, it's not nearly as widely used as Python. Stack Overflow's 2019 developer survey found that 41.7% of respondents use Python, compared to just 8.2% who use Go.⁹⁰

Therefore, we might expect that the total number of potential contributors for a Go-related project would be smaller than that for Python.

Taken together, these factors can help us identify what we mean by “big” or “small” open source projects. Bootstrap is a “big” project in terms of *user adoption*, but is “smaller” in terms of *technical scope*, so it’s unsurprising that only two developers, mdo and XhmikosR, account for most of its commits.⁹¹ Bootstrap does experience moderately high issue volume (or *support required*), so another developer, Johann-S, is more active in responding to Bootstrap’s open issues, while making fewer commits.

These factors can also affect projects in unintuitive ways. There are differences between how a project attracts new contributors and how it retains existing contributors. High barriers to entry, for example, might mean that those developers who make it through the gauntlet will feel a higher affinity to the project.

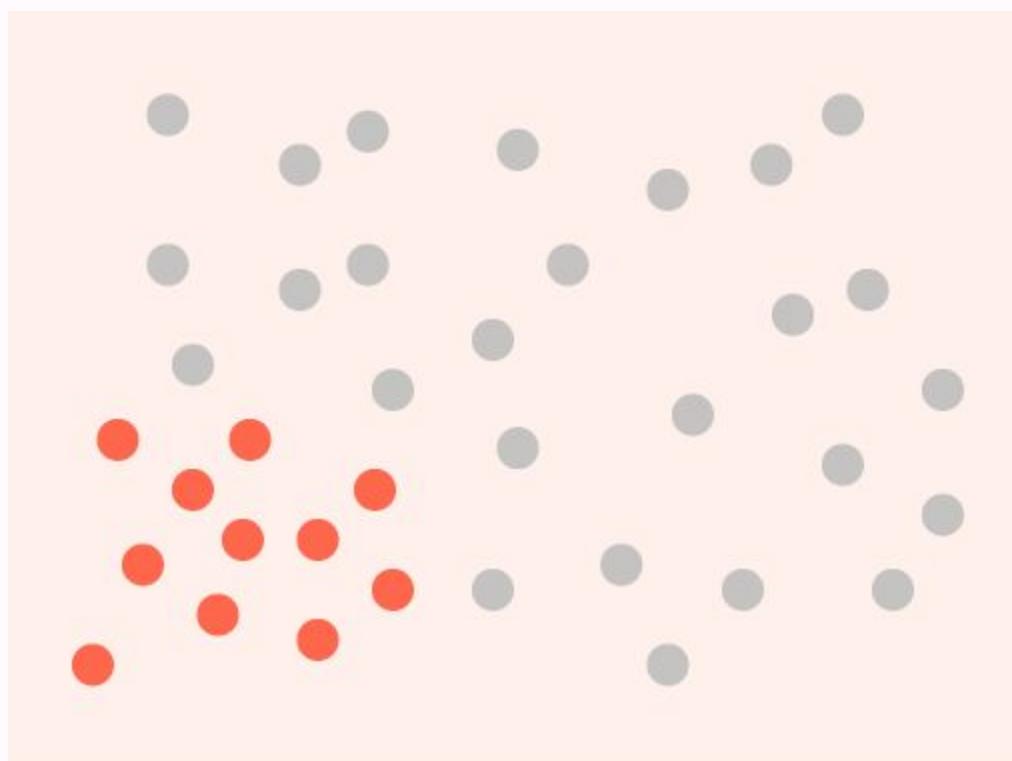
Older projects with messy codebases can, strangely, engender a small but very dedicated group of contributors. The same goes for tooling: I’ve heard more than one Go developer confess they like that Go uses Gerrit instead of GitHub because this cuts down on the noise. On the other end, projects with widespread user adoption can create a sort of bystander effect: nobody contributes because they assume someone else is probably doing it. How these factors affect contributor growth depends on the project in question, but it’s important to acknowledge that they do affect projects in some way.

Focusing on the relationship between contributors and users, we can think of projects in terms of their *contributor growth* and *user growth*. This gives us four production models: *federations*, *clubs*, *toys*, and *stadiums*.

	HIGH USER GROWTH	LOW USER GROWTH
HIGH CONTRIBUTOR GROWTH	Federations (e.g., Rust)	Clubs (e.g., Astropy)
LOW CONTRIBUTOR GROWTH	Stadiums (e.g., Babel)	Toys (e.g., ssh-chat)

Various types of open source projects, classified by user and contributor growth.

FEDERATIONS



● contributors ● users

Federations are projects with high contributor growth *and* high user growth. These are the “bazaars,” first described by Eric S. Raymond, which we typically think of when we imagine an open source project. These projects are rare but impactful: just as most startups don’t end up like Facebook, most open source projects aren’t Linux. Although they comprise a small percentage of open source projects (less than 3%, according to one study), federations occupy the most mindshare due to the size of each project.⁹² Rust, Node.js, and Linux are all examples of federations.

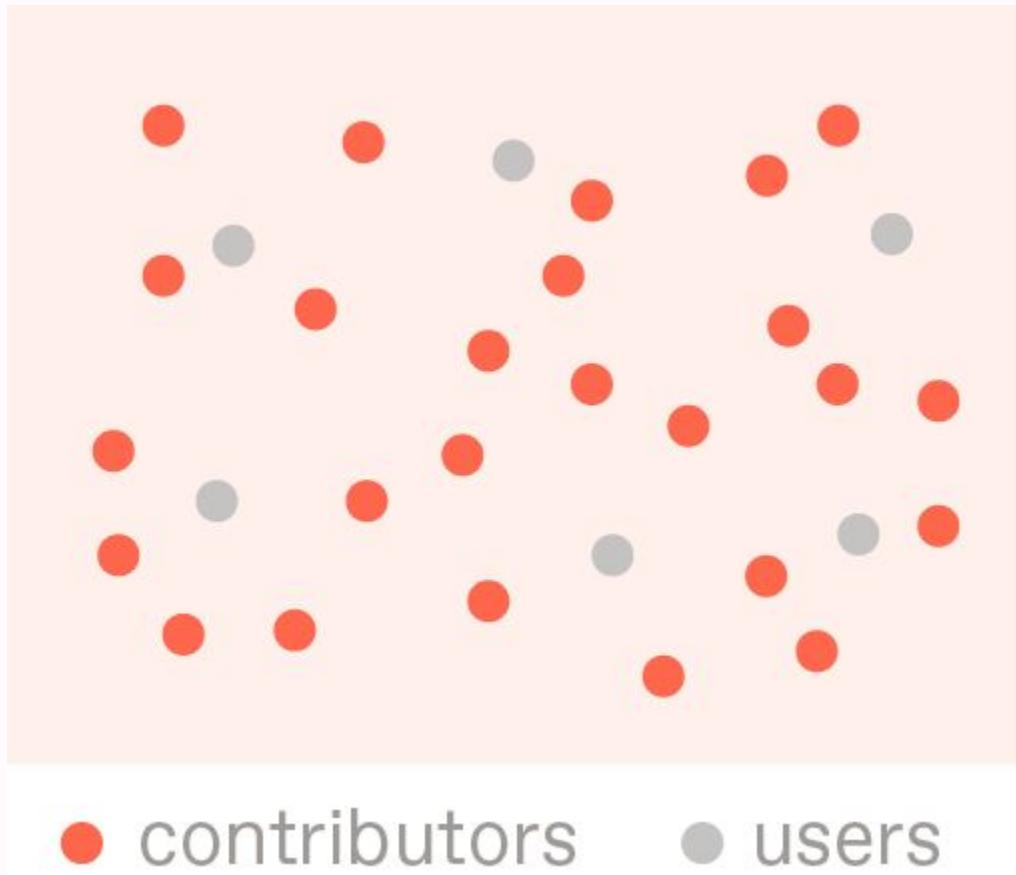
Federations are similar to companies or NGOs. They’re more complex to manage from a governance standpoint, so they tend to develop processes—voting, leadership positions, foundations, working groups, and technical councils—that address coordination issues within their contributor community. These contributors, in turn, make decisions for a broader constituency of passive users.

As their contributor community grows, federations typically “shard” contributors into smaller working groups, where maintainers specialize in certain areas of a project, such as infrastructure or community. Federations also often employ an RFC (request for comments) process, similar to a ballot initiative, to manage major change proposals to the project.

Eventually, however, these working groups can experience the same bottleneck issues that the entire project did beforehand, with a lead maintainer serving as a point of failure for their subdomain. So some federations, such as Node.js, also experiment with liberal contribution policies,⁹³ or what open source developer Pieter Hintjens once dubbed “optimistic merging.”⁹⁴ Instead of gatekeeping new contributions, these maintainers try to distribute ownership more widely, which encourages

more people to become active contributors. Rather than try to contain enthusiastic developers, liberal contribution policies give them a path to spread and grow—like a forest fire.

CLUBS



Clubs are projects with high contributor growth and low user growth, leading to a roughly overlapping group of contributors and users. While there are fewer users overall, these users are more likely to participate as contributors. Astropy, for example, is a package that provides core functionality and tooling for those using Python for astronomy and astrophysics. While Astropy will never be used by most developers, the package's narrow focus makes it easier to recruit contributors and maintain relevance among those for whom Astropy is extremely important.⁹⁵

Programming languages like Clojure, Haskell, and Erlang are not nearly as widely used as Java, C++, and Python. But these languages are useful in certain niches, whether mathematics or telecommunications. (One developer told me, somewhat tongue in cheek, that knowing Haskell means you'll always have a job, because companies that hire for Haskell developers are thrilled to find anyone who can write Haskell at all.)

Clubs are similar to meetup or hobby groups: they attract a narrow group of users, who then also become contributors because they have higher context for the project's activity and feel a sense of affinity to the group. Clubs may not have a wide reach, but they're loved and built by a group of enthusiasts.

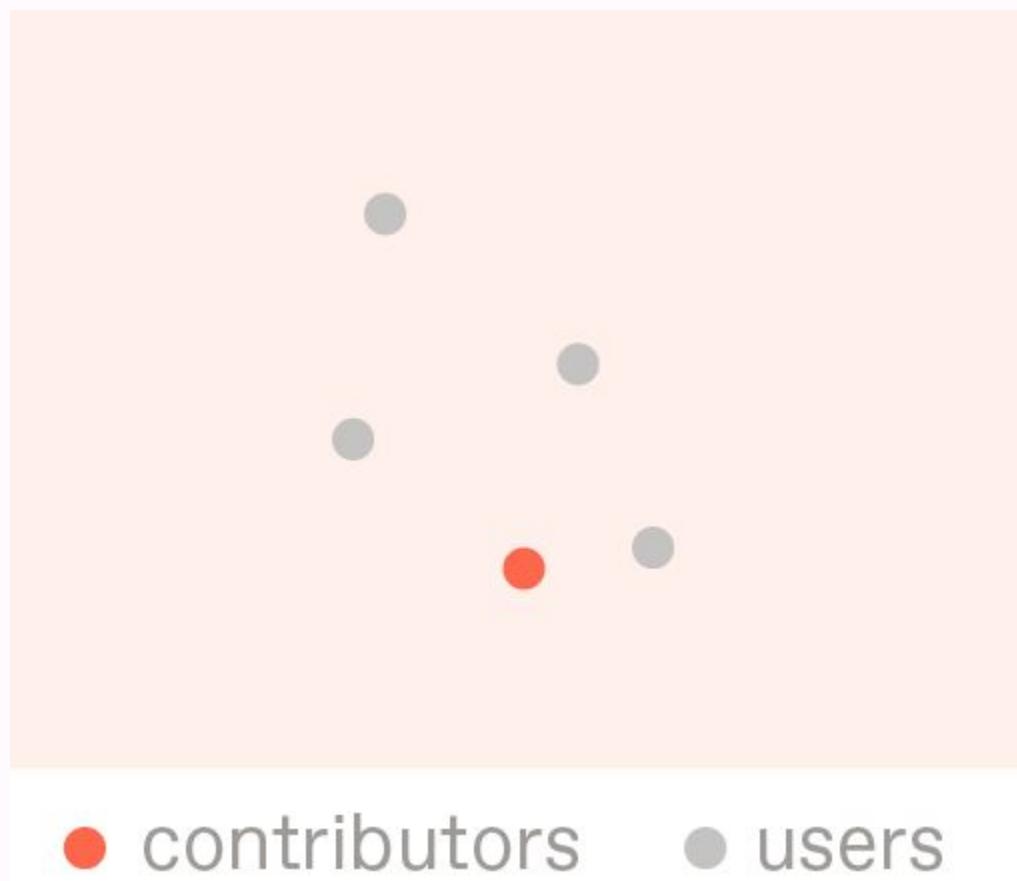
So long as a small group of developers continues to use and contribute to a project, that project can survive indefinitely, regardless of how many users it has.[§] Like obscure message boards, or online communities that are well past their heyday but still survive, clubs tend to be stable, so long as there are enough newcomers who join as older regulars fade away.

Kazuhiro Yamashita et al. describe contributor retention using the terms “magnet” and “sticky,” which were originally coined by the Pew Research Center to describe population migration trends.⁹⁶ Magnetic projects are those that *attract* a large proportion of new contributors. Sticky projects are those where a large proportion of contributors *continue* to make contributions.

Successful clubs are highly sticky, retaining a large portion of their contributor base, even if they don't attract as many new ones. So long as those contributors stick around, and clubs add just enough new contributors to stay active, these projects will continue to exist as their own, self-sustaining commons.

Like federations, clubs attract new members, but they tend to be more selective, expecting higher levels of activity from their participants. In a club, most users are contributors: you're either participating or you're not in the club. Because there's a smaller community overall, who joins the "club" matters more than in a federation, where most users aren't contributors and there's plenty of work to go around. It's the difference between living in a small town and a big city: in a city, communities easily divide into smaller groups, but in a small town everybody knows everybody else's business and cares more about who's spending time with whom.

TOYS

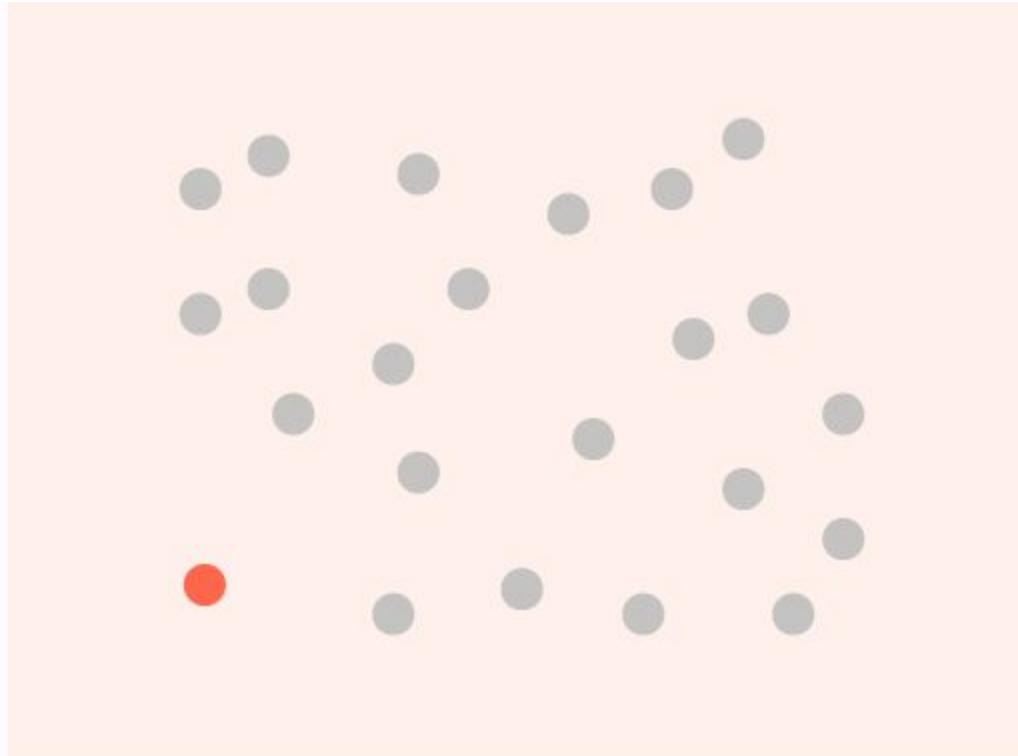


Toys are projects with low contributor growth and low user growth. They're probably the least interesting production model to analyze for the purposes

of this book, because they are effectively personal projects. Toys are like a side project or a weekend project. Eventually, they might become more widely used, but in their current stage they’re just something that an individual developer enjoys tinkering around with for fun. For example, developer Andrey Petrov made a project called ssh-chat, a client that lets users chat through the Secure Shell (SSH) protocol. While the project has thousands of stars on GitHub, it’s meant to be a fun experiment that doesn’t require much upkeep and isn’t trying to grow its user base.^{[98](#)}

Open source projects on GitHub with fewer than ten stars would also fall into the toy category. (While stars aren’t a perfect measure of popularity, they can give some indication of how many people have come across the project.) These projects might have an open source license, but at this stage their authors don’t expect to receive many contributions, nor do they assume anyone is even watching what they’re doing.

STADIUMS



● contributors ● users

Stadiums are projects with low contributor growth and high user growth. While they may receive casual contributions, their regular contributor base does not grow proportionally to their users. As a result, they tend to be powered by one or a few developers. Many widely depended-upon packages and libraries fit into this model, including webpack, Babel, Bundler, and RSpec. Stadiums are becoming increasingly commonplace today.

In a stadium model, one or a few maintainers make decisions on behalf of a broader user base. Unlike a federation or club, whose communities are *decentralized*, a stadium's community assumes a *centralized* structure, centered around its maintainers. Decentralized communities are defined by

many-to-many interactions, but centralized communities have a *one-to-many* social structure.

Anthropologist Spencer Heath MacCallum calls these *proprietary communities*, where every member “is related to the whole organization through its proprietors by contract.”⁹⁹ (“Proprietary,” in this context, does not refer to a community’s commercial nature, but rather to the idea that it has a “proprietor,” such as an owner or trustee, through which all other community interactions flow.) Hotels, airplanes, and mobile home parks are all examples of proprietary communities in the physical world. The guests, patrons, and residents of these communities are related to one another through a central proprietor: the hotel owner, the airline, the park manager.

Similarly, in the online world, centralized communities tend to form around individual creators, who “[facilitate] every activity represented in the community by the act of providing space for it,” and who are thus tasked with maintaining its function.¹⁰⁰ The community exists because the creator has provided something for people to gather around. One esports player highlights how this phenomenon occurs on Twitch, noting that “every streamer has a mini community who watch them and who like each other because they’re attracted to basically the same person.”¹⁰¹

In comparison to decentralized communities, centralized ones don’t rely as strongly on defined governance processes, because social interactions primarily take place not among regular contributors but between a creator and their users. It’s the creator, rather than their users, who is “responsible for the basic economic structure of the community.”¹⁰²

Number of Commits, Top 100 Contributors, Rust vs. Clojure



Distribution of commits among the top hundred contributors to Rust vs. Clojure. [103](#)

The lopsided distribution of work in a stadium model can be partly explained by *supply-side economies of scale*. Software, like physical infrastructure (roads, utilities, telecom), has high fixed costs of initial production, followed by low marginal costs. In other words, it's expensive to get started, but each additional user incurs relatively low costs.

An electricity company has high up-front costs, but once its infrastructure is in place each additional customer is cheap to serve. As a result, these types of industries tend to become monopolies, due to economies of scale. It's just cheaper to consolidate those high fixed costs in one place, and it's harder for newer entrants to overcome them.

Similarly, if we think about who supplies labor to an open source project, it's "expensive" to onboard new maintainers, because maintenance often

requires knowledge that isn't easily externalized to others. So newcomers tend to make casual contributions, instead of pitching in on more complex tasks around project management.

Given the high fixed costs of entry into maintainership, the knowledge required to maintain tends to stay concentrated among one or a few people. The longer they go without externalizing this knowledge, the more difficult it becomes for newcomers to participate.

Open source projects can move between quadrants. Early-stage projects that start out as clubs might mature into federations as their user growth rates change, just as toys might mature into stadiums. Other projects might have high user growth but artificially low contributor growth due to factors such as poor contribution policies. By making it easier to contribute, or reducing technical scope, maintainers might move their projects from a stadium to a federation model.

Decentralized communities, e.g., clubs and federations, are characterized by the potential for high contributor growth. The future of these projects hinges upon recruiting new contributors, as well as reducing friction to contribute. Mikeal Rogers, who worked on Node.js in the early days, describes this model in his blog post “Healthy Open Source”:

The purpose of [Node.js’s contribution] policy is to gain contributors, to retain them as much as possible, and to use a much larger and growing contributor base to manage the corresponding influx of contributions. . . .

Avoid creating big decision hierarchies. Instead, invest in a broad, growing and empowered contributorship that can make progress without

intervention.¹⁰⁴

By contrast, centralized communities operate on the basis of limited attention. As proprietors of their communities, creators must manage user demand on their own. As a result, they tend to rely more heavily on automation, distributed user-to-user support, and the aggressive elimination of noise. While all popular projects utilize these approaches, they're a necessity for stadiums.

This characteristic of stadiums, versus clubs and federations, also highlights the critical role of platforms in enabling centralized communities.

Decentralized communities, whether big online forums or small group chats, will always be more agnostic as to where they till their land. If times get tough, it's easier for them to pack up camp and migrate to greener pastures. Jon Hendren, who helped run the popular online forum Something Awful, recalls that, as traffic waned to their site, "most of us moved to Twitter or whatever."¹⁰⁵ However, he says, "We kind of brought the community with us. . . . I'm in a group DM with most of the same guys I've been talking to for the past 12 years now. The community isn't dead, it was just moved, mostly."

Similarly, What Would Virginia Woolf Do?, a community for women over forty, grew to over 30,000 members as a Facebook group before moving to its own platform. For a community of this size, Facebook became more of a nuisance than a lifeline. The group's founder, Nina Lorez Collins, explained to a reporter that "Facebook is a flat landscape, so managing 38 [sub]groups —subject, regional, and admin—on Facebook was a bear, and they gave us no support whatsoever."¹⁰⁶ Collins decided that "the only way forward was to go with a subscription model and build a branded app [where] we could

. . . finally control our own universe.” Creators, on the other hand, are tethered to the support that platforms provide, because platforms significantly reduce their costs, enabling them to accomplish far more than they could have on their own.

The difference between these project models is not precisely defined. At either extreme, we can see clear differences. We can say that Linux, a project with multiple subprojects and working groups, does not look like tslib, a small helper library for the programming language TypeScript. But the middle is a little murky. There are some projects with four or five maintainers, and other projects with many contributors who are not deeply involved.

Different project types carry different implications for how to effectively manage contributor communities. The writer Kevin Simler, examining the differences between open and gated communities, explains that, “If you’re organizing a flea market, it would be counter-productive (not to mention insane) to require background checks and letters of reference from all buyers and sellers. But if you’re running a diamond dealership, maybe those precautions are necessary.”¹⁰⁷ By identifying these extremes, we can better understand how to think about the middle.

* According to Wikipedia, “*Ptychobranchus subtentum*, also known as the fluted kidneyshell, is a species of freshwater mussel, an aquatic bivalve mollusk in the family Unionidae, the river mussels.”⁶⁵

† One person I spoke to compared the Drupal project to the Basques as a means of describing Drupal’s culture and history. Like the Basque Autonomous Community, Drupal is sequestered in its own territory; not being on GitHub led its developers to develop their own norms,

which are influenced by, but distinct from, mainstream open source practices.

‡ Thanks to Devon Zuegel for this analogy.

§ In her 2019 *Gizmodo* article “The Gentle Side of Twitch,” Nicole Carpenter similarly describes a subset of Twitch streamers who have smaller but highly engaged audiences. One of the streamers she interviews is Jennifer Chambers, who streams herself knitting to a “modest, but steady group of viewers”: “Streaming on Twitch, for Chambers, is less like a musician performing to a stadium-full of fans and more like a knitting club.”⁹⁷

ROLES, INCENTIVES, AND RELATION- SHIPS

03

“Til recently you were online for a reason. I’ve met people in cafés, but never had to leave one because 3 billion people suddenly walked in.”

—STAR SIMPSON, via Twitter¹⁰⁸

While it feels obvious today that we want to freely share the things we make, the early success of open source captivated scholars and economists because it defied everything we thought we knew about how and why people create.

Open source developers were frequently characterized as “hobby” developers (most famously in Bill Gates’s 1976 “Open Letter to Hobbyists,” which we’ll get to later), because the assumption was that only companies could make “real” software. Even Carl Shapiro and Hal R. Varian’s *Information Rules: A Strategic Guide to the Network Economy*, a 1999 book widely regarded as the definitive text on the economics of information goods, hardly gives open source software a glance, instead treating software as a commodity to be bought and sold by companies.

Once companies started using open source for commercial purposes, and people realized that these “hobby projects” were able to compete with the software made by paid employees, scholars had to come up with a new framework to explain this behavior.

Previously, our understanding of how and why people make things was modeled after Ronald Coase’s theory of the firm, which proposes that *firms* (i.e., companies, organizations, and other institutions with centralized resources) naturally emerge as a way to reduce transaction costs in the market.¹⁰⁹ Coase would’ve told us that only companies make software because, from a coordination standpoint, managing the resources required

to pull off such a feat would be most efficiently handled within the same organization.

By contrast, the open source projects attracting attention in the late 1990s and early 2000s—the Linux kernel, which powers operating systems; Apache, an HTTP server; FreeBSD, an operating system; GNOME, a desktop environment—were produced by distributed groups of developers that transcended employer affiliations.

Coase’s theory of the firm fails to explain why these developers would find one another and make software together, despite a lack of both formal contracts and financial compensation. In terms of transaction costs, collaborating on open source software with unaffiliated individuals should be too “expensive,” compared to writing software with one’s coworkers.

But a few people noticed that these open source projects operated like communities, so they instead explained the projects’ behavior by describing them as a *commons*, meaning a resource that is owned, used, and managed by a community. These communities rely upon self-governed rules, rather than outside intervention, to manage the resource and avoid over-provisioning or depletion.

A THEORY OF THE COMMONS

In the second half of the twentieth century, the economist Elinor Ostrom spent decades studying the conditions that lead to a flourishing commons, such as forests, fisheries, irrigation systems, and other common pool resources.* She tried to understand how people produce in a commons, and why some resources are successfully self-managed, thus avoiding the so-called “tragedy of the commons” (wherein resources are depleted by people acting in their own self-interest, rather than in the collective interest) and the need for either market or government intervention.

Through her research, Ostrom identified eight design principles that contribute to a well-managed, successful commons:

1. Membership boundaries are clearly defined.
2. The rules that govern the commons should match the actual conditions.
3. Those who are affected by these rules can participate in modifying them.
4. Those who monitor the rules are either community members or are accountable to the community, rather than outsiders.
5. Those who violate the rules are subject to *graduated sanctions*, which vary depending on the seriousness and context of the offense.
6. Conflicts should be resolved within the community, using low-cost methods.
7. External authorities recognize the right of community members to devise their own institutions.
8. If the commons is part of a larger system, its governing rules are organized into multiple “nested” layers of authority.¹¹⁰

Thematically, these conditions point to the need for a strong sense of group identity, which makes governance processes like rules, dispute resolution, and sanctions (i.e., corrective actions) more meaningful.

When the boundaries of the community are clearly defined, members know who belongs and who doesn't. They write the rules that govern their own community. And they have *high context* for one another's actions, which fosters mutual *trust*.

Members also have a *low discount rate*, which is another way of saying they have “skin in the game,” meaning they intend to participate in the community for a while. This means that sanctions, and even the threat of sanctions, help set social norms effectively, because members care about not getting kicked out of the community. When someone misbehaves, they may be punished by an “official” moderator, but, even more importantly, they are swiftly shamed by their peers, which serves as an unofficial form of moderation.

A low discount rate also means that members are biased toward cooperation. Like being trapped in an elevator with strangers, if members are all stuck with one another for a while, they’re more inclined to figure out strategies to make things work, such as developing governance processes to handle future disputes.

Like the fisheries in Ostrom’s case studies, online communities can historically be understood as a group of self-organized but disconnected villages. Ostrom spent the latter part of her career applying the principles she had observed to the “digital commons,” where knowledge is the shared resource.

For example, an online forum has regular members, who are distinct from first-timers, lurkers, or casual posters. These members have developed a set of social norms that may not be clearly visible to newcomers, but which are known and enforced among the core group.

Early online forums like Usenet and MetaFilter relied upon self-defined social norms to manage community behavior. Contemporary examples of online communities might include Reddit or Facebook groups, where each subreddit or group has a distinct identity and culture. While certain members occupy distinct roles, the primary mode of social interaction in these communities is distributed, or *many-to-many*.

WHY WE PARTICIPATE IN THE COMMONS

Ostrom's work on the commons helps us understand the conditions in which people produce software *collaboratively*: the clubs and federations of open source. In the early 2000s, Yochai Benkler expanded upon Ostrom's model by applying her findings to the online world. He terms this communal structure *commons-based peer production* (CBPP) in a 2002 essay called "Coase's Penguin, Or, Linux and 'The Nature of the Firm.'" (The title is a reference to Linux's mascot, which is a penguin; in the paper, Benkler leans heavily upon the example of open source software to make his case.)

Benkler observed that people were collaborating online for seemingly no obvious reason beyond personal satisfaction. He tried to understand how and why people would do this outside of Coase's firm (i.e., in their spare time), given that it should be more transactionally expensive.

If Ostrom gives us an organizational theory of the commons, Benkler helps us understand why individual members participate. Benkler reasons that if individuals are personally motivated to do something, coordination costs will be lower.¹¹¹ Unlike companies—which need to solicit, evaluate, hire, and manage employees—the members of a commons simply self-organize based on who wants to do the work most.

Additionally, at a company, only employees can do the work, limited by their job function. But in a commons, anyone can stumble upon an advertised task and volunteer themselves. By removing “property and contract,” the commons will theoretically select for the best person for the job at a lower cost.

Imagine an icebreaker game in which a group of strangers must line themselves up based on each person’s birthdays, from January to December. How do they do this quickly? One strategy might be to have everyone write down their name and birthday on a single piece of paper, then choose a designated leader to read names off the sheet and assign each person to the right place. But the more common outcome is that everyone starts to take charge. One person shouts, “Januarys, over here!,” while another raises their hand for the March birthdays. Once everyone is clustered by month, the subgroups organize themselves by day, before finally adding their sections back into the main group.

Coase’s theory of the firm looks more like the former approach, while Benkler’s commons-based peer production resembles the latter. In terms of coordination effort, it *could* be less costly to designate one person as the leader, whose job it is to collect information from everyone and keep all the work in one place. But when you have a group of strangers who are loosely

affiliated with one another and excited to participate, and nobody is officially in charge, it's more likely that a bunch of people will volunteer all at once. Intrinsic motivation makes it easier for people to self-organize to achieve the same outcome.

Benkler is careful to emphasize that he does not think that commons-based peer production is always preferable to the firm, but rather yet another possible outcome:

I am not suggesting that peer production will supplant markets or firms. I am not suggesting that it is always the more efficient model of production for information and culture. What I am saying is that this emerging third model is (a) distinct from [markets and firms], and (b) has certain systematic advantages over the other two in identifying and allocating human capital/creativity.¹¹²

A few of the conditions that Benkler identifies as necessary to pull off commons-based peer production are *intrinsic motivation*, *modular* and *granular* tasks, and *low coordination costs*.

Intrinsic motivation is the currency of the commons: members do the work because they want to do it. In the case of open source, it's assumed that developers participate because they enjoy writing code.

Guido van Rossum, for example, wrote the programming language Python while looking for a “‘hobby’ programming project that would keep me occupied during the week around Christmas.”¹¹³ And Linus Torvalds released the Linux kernel and operating system as “just a hobby, won’t be big and professional,”¹¹⁴ then released the version control system Git as “some scripts to try to track things a whole lot faster.”¹¹⁵

Benkler proposes that, in order to keep people motivated, tasks must be modular and granular:

When a project of any size is broken up into little pieces, each of which can be performed by an individual in a short amount of time, the motivation necessary to get any given individual to contribute need only be very small.¹¹⁶

Modularity refers to how the project is organized. Can it be broken into clear subcomponents, and do they fit easily back together? **Granularity** refers to the size of each module. It should be easy for anyone to jump in and complete the task without too much preexisting knowledge.

The modular, granular approach to software is embodied by the Unix philosophy, originating from the developers of the Unix operating system, which heavily influenced the design of open source software. As Doug McIlroy, one of its developers, counsels, “Write programs that do one thing and do it well. Write programs to work together.”¹¹⁷

Finally, Benkler suggests that **low coordination costs** are necessary to produce in a commons. In open source, coordination costs include both “quality control over the modules” (such as reviewing code) and “integrating the contributions into the finished product” (such as merging pull requests).¹¹⁸

Coordination work is expensive because it’s not intrinsically motivated. (For example, developers tend to be more excited about writing code than reviewing someone else’s contribution.) And, as anyone who’s tried to delegate work has probably noticed, it’s usually faster to do things yourself than to train someone else to do it.

A maintainer's biggest coordination costs come from reviewing and merging new contributions, so there's an incentive to keep these costs low. When the costs of coordination outpace the benefits, the commons breaks down as a useful production model.

The theory of the commons helps explain the curious behaviors of developers, seemingly unmotivated by money, which led to the early success of open source. It explains the success of prominent open source projects built by big, decentralized communities, like the web application framework Ruby on Rails.

Commons-based peer production also explains why some developers hold the view that money and open source don't mix. If production runs on intrinsic motivation, money is an *extrinsic* motivator that is thought to interfere with an already well-coordinated system. Although the commons might not be as *profitable* as the firm, it's also more resilient, because the currency of its transactions is the desire to participate, rather than money.

David Heinemeier Hansson, who created Ruby on Rails, is a vocal advocate for a commons-based approach to open source production:

External, expected rewards diminish the intrinsic motivation of the fundraising open-source contributor. It risks transporting a community of peers into a transactional terminal. And that buyer-seller frame detracts from the magic that is peer-collaborators.[119](#)

But today, not every open source project looks like a commons. Stadiums have one or a few maintainers, who are surrounded by casual contributors and users. This structure engenders one-to-many, rather than many-to-many, activity. Much of our online behavior today increasingly resembles these

centralized communities, with just one or a few creators who attract a bigger crowd.

Without the safety net of the commons, stadiums need to organize their work differently. Decentralized communities prioritize work based on *abundance* of attention: encouraging new contributors, developing governance processes, and improving engagement and retention. But a creator prioritizes work based on *scarcity* of attention: saying no to contributions, closing out issues, reducing user support. While the commons is tasked with resolving *coordination* issues, creators are defined by the need for *curation*.

Josh Lerner and Jean Tirole's "The Simple Economics of Open Source" was published in 2000, but it contains analysis and observations that are still trenchant today. The widely cited paper questions whether the commons was ever tractable in open source, or just a temporary phenomenon:

Can the management of open source projects accommodate the increasing number of contributors? The frequency and quality of contributions to each of the open source projects studied appears to be highly skewed, with a few individuals (or at most a few dozen) accounting for a disproportionate amount of the contributions, with most programmers making just one or two submissions. . . . If large numbers of low-quality contributions are becoming increasingly common, there may be substantial management challenges in the future.[120](#)

Why are we seeing more centralized communities today? Platforms accelerated this transition by making it easier for people to move between

communities, which made collective identity more porous. In the case of open source, that platform was GitHub.

HOW PLATFORMS BROKE APART THE COMMONS

Before open source moved to GitHub, each project could be thought of as a commons. While there might be a lot of activity *within* a project, there wasn't a clear relationship *between* projects.

As with a commons, a project's members felt a shared sense of ownership over the code, and users were treated as potential contributors. If you found a problem, you were expected to roll up your sleeves and fix it.

GitHub was the highway system that transformed how open source software was produced. While every project still has its own social and technical norms, the barrier to contributing to an unfamiliar project is much lower than before. At some level, every project on GitHub now looks the same, regardless of its language or function. Each project has a landing page with a README, a place to download the code, an issue tracker, and a list of pull requests.

While some like to grumble at GitHub's homogenizing effects, what happened in open source isn't much different from what happened to the rest of the internet. Before platforms, our online world was a scattered collection of forums, blogs, personal websites. People might have felt an affinity to, say, a particular message board, but they had little sense of who else was out there, and they largely stayed out of one another's way.

It was only in 2006, when Facebook introduced the News Feed, that we got our highway system. Roads and bridges, like hyperlinks and news feeds,

brought together previously disconnected rural towns, forging paths between those that once sat blindly within arm's reach. Blogger Eugene Wei describes the impact of the News Feed as such:

In the annals of tech, and perhaps the world, the event that created the greatest social capital boom in history was the launch of Facebook's News Feed. . . .

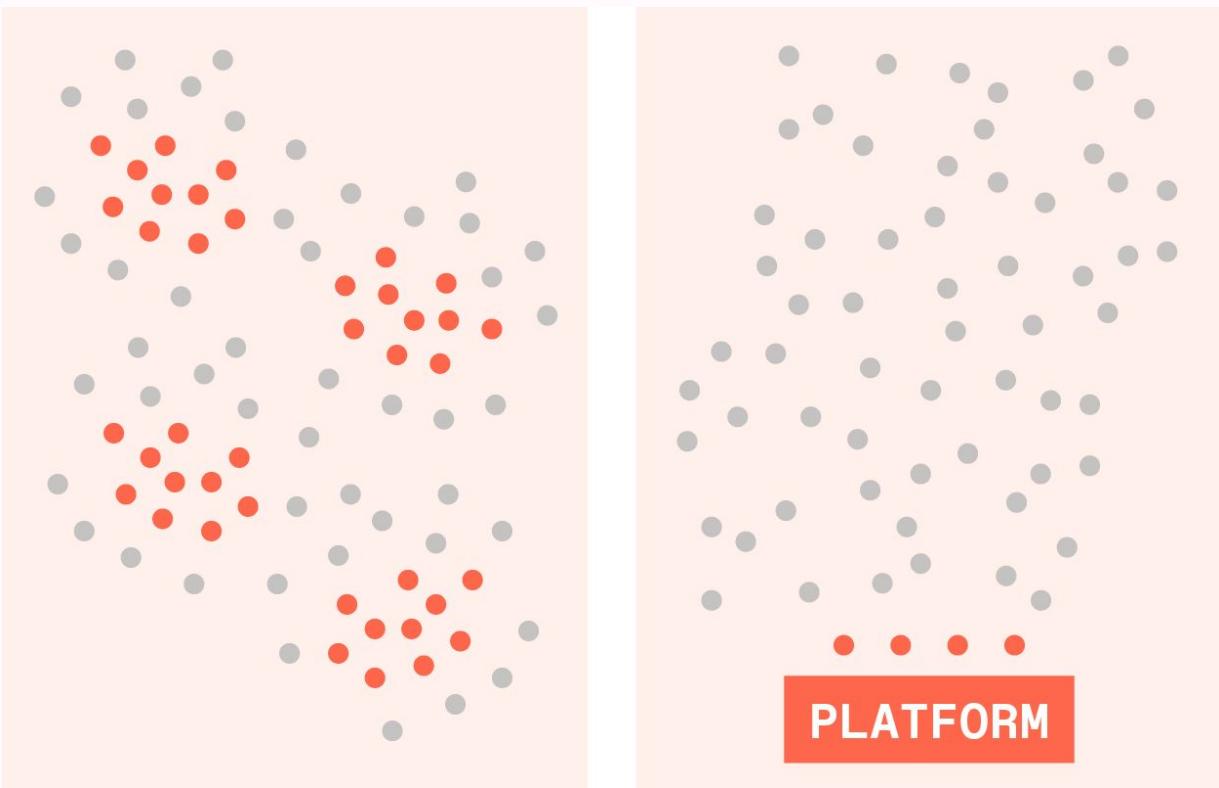
By merging all updates from all the accounts you followed into a single continuous surface and having that serve as the default screen, Facebook News Feed simultaneously increased the efficiency of distribution of new posts and pitted all such posts against each other in what was effectively a single giant attention arena, complete with live updating scoreboards on each post. It was as if the panopticon inverted itself overnight, as if a giant spotlight turned on and suddenly all of us performing on Facebook for approval realized we were all in the same auditorium, on one large, connected infinite stage, singing karaoke to the same audience at the same time.^{[121](#)}

By connecting towns to one another, highways change communities' underlying social structure. Highways enable migration and cross-pollination of ideas. Without highways, residents tend to stay in the towns they grew up in. When these pathways are opened, collective identity is eroded. Instead of "competing for status in small tribes," people find themselves "dropped into a talent show," competing against strangers.

Anthropologist Michael Wesch uses the term "context collapse" to describe how YouTube's wide reach affects how people present themselves online. Instead of having in-person interactions, which occur within a specific

context, YouTube creators experience “an infinite number of contexts collapsing upon one another into that single moment of recording.”¹²²

While Wesch focuses on the effects of context collapse on *individuals*, online communities experienced their own version of context collapse when platforms turned the “panopticon” upon itself, inducing a flood of newcomers, spectators, and drive-by visitors. The collective identity of communities was reduced in favor of personal identity, with individuals acting as free agents.



Breaking up the commons: from decentralized communities to nested platforms.

Today, most developers transact casually with one another, across projects, with low context and little skin in the game. These developments present a challenge to Ostrom’s definition of a commons, and, by extension, to our

current working theories about how and why open source software is produced.

In order for sanctions to be effective, for example, communities need cohesion. But if developers don't see themselves as "members" of a community, they might not necessarily mind being shamed or kicked out, and may indeed even thrive on such conflict.

Researchers Robert E. Kraut and Paul Resnick, who did extensive early research on online communities, observe that "people sanction misbehaviors because in the long run, doing so improves the welfare of the groups of which they are a part."¹²³ But "trolls and manipulators are outsiders who have no vested interest in the community functioning well. This problem is especially difficult to deal with because social sanctions . . . may either have no effect, or, in the case of trolls, may actually increase their activity."¹²⁴

But defining membership boundaries in an open source project turns out to be one of the hardest problems to solve. We can see this difficulty expressed in the task of implementing and enforcing *codes of conduct*: policies that govern acceptable behavior for a community.

In recent years, it's become more common for open source projects to have codes of conduct. This decision sometimes leads to emotionally charged discussions between those who see codes of conduct as politically motivated and others who don't understand how enforcing civil behavior could possibly be controversial. At the heart of these conversations lies a question of who has the authority to make and set policies for a given community.

One prominent controversy centered around a code of conduct took place in 2015 on Opal, a Ruby-to-JavaScript compiler project, in an episode later dubbed “Opalgate.” After one of Opal’s maintainers made a comment about transgender people on Twitter, his comments prompted a Ruby developer to open an issue on Opal with the subject line “Transphobic maintainer should be removed from project.” The developer linked to the tweet, asking, “Is this what the other maintainers want to be reflected in the project? Will any transgender developers feel comfortable contributing?”^{[125](#)}

Another maintainer promptly closed the issue and shot back,

If you want him removed, start working on Opal and contributing as much as him to everything he did for Opal so we have a replacement that’s more in orientation with your morals and views.

Protip: you won’t because you can’t.

The issue thread became a battleground, with outside developers pouring in to add their support for either side, until another maintainer locked the issue a few days later, preventing further comments. The conversation was redirected into another long issue thread about adding a code of conduct to Opal, which was ultimately adopted.^{[126](#)}

Although Opalgate was ostensibly about whether a maintainer’s personal political views should affect their ability to participate in the project, another reason the issue blew up the way it did is because it was opened by, and attracted pile-on from, developers *outside* of Opal’s contributor community. But what defines an outsider in open source, when any developer is supposed to be able to participate? If everybody is a potential contributor, who gets to make, enforce, and follow the rules?

Kraut and Resnick observe that a community needs to “protect itself from the potentially damaging actions” of newcomers in order to survive,^{[127](#)} since newcomers can destabilize preexisting social norms: “Because newcomers have not yet developed commitment to the group and have not yet learned how the group operates, it is rational for established group members to distrust them.”^{[128](#)}

The newcomer effect is also known as the “Eternal September” problem, a term coined by members of the early online community Usenet, which experienced an influx of newcomers every September due to new students getting access for the first time. But once America Online (a sort of early highway system itself) began offering access to Usenet, the service provider exposed the community to a constant stream of new users, creating an “eternal September.”

Kraut and Resnick further suggest that successful online communities need to “designate formal sanctioning rules so that those imposing sanctions have legitimacy,” noting that this role typically falls upon moderators or administrators, who are “less likely to generate drama or retaliation than the same message coming from someone without a formal role.”^{[129](#)} Today, however, these dynamics seem to be reversed, if anything. A maintainer who attempts to assert their authority will risk *more* wrath and ire, as evinced in the case of Opalgate, by inadvertently creating a beacon that attracts more outsiders to their community.

When the issue is about the maintainers themselves, it’s particularly difficult to determine who should have governing authority. Do maintainers get the final say, or should communities be able to influence the outcome?

Deferring to one's community seems to be more obviously aligned with democratic values, but open source's distributed nature makes that ideal challenging to live up to. If maintainers were to defer to their community, how could they assess overall opinion, given the lack of clear membership boundaries? Countries have citizenships and constituencies, but open source projects are open to anyone. If someone made a one-time contribution to Opal, does that make them a “contributor” in the same sense as a lead maintainer? How do we know if the developers opposing or favoring action represent a minority voice, if we don't know the actual population of the community?

The former option—deferring to maintainers—carries its own set of challenges, as maintainers don't always agree with one another's decisions. In 2018, a maintainer of Lerna, a tool for managing JavaScript projects, decided to add a clause to its MIT license “banning ICE collaborators”—that is, companies known to work with the United States Immigration and Customs Enforcement agency. In the pull request, the maintainer explained, “I've been really disturbed to see what ICE has done to American immigrants . . . a lot of big tech companies are supporting ICE by providing them with infrastructure and in some cases doing significant development work for them.”¹³⁰

While the decision had been discussed with two other stakeholders before the pull request was opened, it was not unanimously supported by Lerna's other maintainers, and the support it did receive was not very strong. One developer consulted beforehand was Lerna's author, who clarified in the pull request thread that “I would not have personally made this change. I do however respect the existing maintainers of the projects [sic] decision to do so. I do not consider this project to be mine.” The other maintainer who was

consulted reverted the change the next day and apologized to Lerna’s community, explaining that “despite the most noble of intentions, it is clear to me now that the impact of this change was almost 100% negative.”¹³¹ Lerna’s maintainers also decided to remove the maintainer who proposed the license change, as “it has been very clear for quite some time now that he has decided to cease making constructive contributions.”

Thus, even within a group of people who could ostensibly be called “Lerna maintainers,” there are unspoken layers of influence. Lerna’s author abdicated his decision-making power. The maintainer who proposed the change did not have sufficient buy-in from his collaborators to pull it off, and was removed from the organization.¹³² Getting weak support from a fellow maintainer was not enough to execute the change.

Finally, those who socially influenced these decisions weren’t just limited to maintainers of Lerna’s project repository. Sean Larkin, a maintainer of webpack, the JavaScript file bundler, commented on the follow-up announcement, “Wonderfully said. We are in this together now as open source maintainers of other projects like webpack and babel, and you don’t have to feel responsible for it all.”

These governance challenges speak to why many of the bigger open source projects prefer to use *consensus-seeking* over *consensus* processes. When it’s unclear where to draw community boundaries, voting systems don’t work very well, because there’s no way to know whether those votes are representative of the total population.

The Internet Engineering Task Force (IETF), which develops the standards behind our internet protocols, uses what it calls “rough consensus” to reconcile the tension between authoritative and democratic governance,

given a hard-to-define constituency: “Our credo is that we don’t let a single individual dictate decisions (a king or president), nor should decisions be made by a vote.”¹³² In a consensus-seeking model, the goal is not to “win” votes or come to a unanimous agreement, but rather to ensure that there’s a forum for people to raise and discuss their concerns, and that nobody feels strongly enough to block the group from moving forward. Consensus-seeking emphasizes discussion over enumeration: “Rough consensus is achieved when all issues are addressed, but not necessarily accommodated.”

Consensus-seeking models work for projects with bigger contributor communities, whether federations or clubs. In these communities, it’s still possible to foster a strong sense of membership, which translates into social norms, rules, and sanctions that newcomers are expected to acquiesce to.

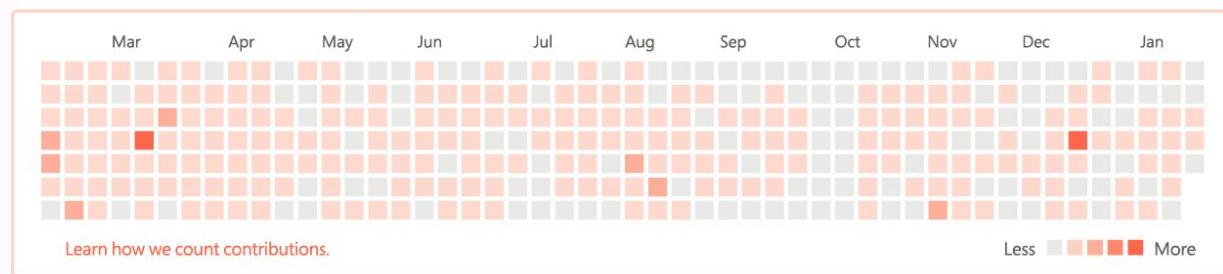
But centralized communities, such as stadiums, don’t have active “members” like a decentralized community does. Given that there are fewer people who are actively involved, stadiums are better suited to “benevolent dictator” governance models than consensus-seeking ones.

A NOTE ON TERMINOLOGY

In the sections that follow, I'll examine the roles of *maintainers*, *contributors*, and *users*. None of these terms are easily defined. In particular, there's quite a bit of discussion among open source developers around what constitutes a "contributor."

GitHub defines *contributors* as those who've made commits that were merged into a project's codebase. A project with, say, fifty contributors has received commits from fifty unique developers.^{[133](#)} A user's GitHub contribution graph—a colorful patchwork of green squares, which visualizes their activity across projects—is generated from a mix of making commits, opening pull requests and issues, and submitting pull request reviews.^{[134](#)}

5,067 contributions in the last year



A GitHub contribution graph (shown here in red).^{[135](#)}

But there are plenty of other contributors that get left out: for example, those who triage issues or respond to user questions on mailing lists, both of which are essential contributions to an open source project. Developer Kent C. Dodds proposed a standard called All Contributors for "recognizing contributors to an open source project in a way that rewards each and every contribution," complete with an emoji key to display these types of

contributors in the README file of a project.¹³⁶ Some contributions, like organizing events or providing financial support, tend to happen away from a project’s code repository, and are thus more difficult to measure and track, but socially they are generally considered to be contributions.

The project description for All Contributors reads, “Recognize all contributors, not just the ones who push code.”¹³⁷ As this description suggests, the debate around defining contributors tends to center around “code” versus “non-code” contributions (the implication being that commits are for code). But these terms strike me as a bit of a misnomer, because contributors can make non-code commits to a repository, too. Tasks like writing documentation and fixing broken links, for example, also make their way into projects in the form of commits.

Rather, I suspect we’ve converged upon commits partly because they’re comparatively easy to measure, and partly because we do need a way to talk about a particular subset of the open source workflow—issues, pull requests, code review—as opposed to, say, publishing blog posts or event planning.

It would naturally follow that a person who makes a “contribution” is called a “contributor,” but I think it’s the conflation of these terms that’s led to so much confusion. Being a contributor is a form of identity, regardless of the contributions one has made to a project. Like joining a club, it’s not about how many times you’ve attended meetings but how you self-identify, and how others identify you, that makes you a “member.” Some attendees might come every week for years and still not be considered part of the group.

Whether someone is considered a contributor is a negotiation between the developer in question and the existing contributor community. The same

contribution to two different projects could be perceived very differently. Some projects are more permissive about who they call a member, while others are more conservative. Like any other club or group, what counts as a “leader” or a “member” in one community might be completely different in another. These terms are determined locally, rather than by a generalized definition that works across all projects.

For example, the term “casual contributor” is used to refer to someone who’s made a few commits to the project but isn’t otherwise involved. On a bigger project, getting even one small contribution merged in can be a status symbol. The developer who made the pull request might proudly call themselves a “contributor to Rails,” and it’s understood that this title might be somewhat tongue in cheek. But on smaller projects, casual contributors might just see themselves as users. They feel empowered to make small changes and fixes to the project, but that doesn’t necessarily mean they identify as contributors.

Similarly, those who organize meetups, give talks about the project, or answer questions on Stack Overflow might not see themselves as contributors if they don’t regularly interact with the project’s core developers. When they say they see themselves as part of, say, the “Rust community,” they might primarily be thinking of other developers who *use* Rust, rather than of those who contribute to the Rust project.

Membership is a two-way social contract. Some developers don’t want the responsibilities of being recognized as a contributor. It’s possible they just want to contribute a one-time fix and be on their way, the equivalent of picking up a stray piece of trash on the street. They might prefer the

freedom of making video tutorials or organizing events on their own terms, without the additional overhead of coordinating with other contributors.

The ambiguous definition of a “contributor” can cause confusion between projects. In a club or federation, an active contributor cohort might display prosocial attitudes that lend themselves to making decisions collectively. But that attitude doesn’t necessarily port over to the stadium model, where contributors act more like users, and only the maintainer makes decisions on behalf of a project. (One maintainer, rolling his eyes, showed me a Hacker News thread in which a developer espousing strong opinions claimed to be a “contributor” to his project. That contribution turned out to be a single pull request, from several years ago, which removed a bit of white space.)

The difficulty of distinguishing contributors from users also applies to distinguishing between contributors and maintainers. There is no universal definition of a maintainer. We could say that it has something to do with having commit privileges, or how frequently someone makes commits. But every definition quickly breaks down across projects, because, like the term “contributor,” “maintainer” is an identity more than a function.

Being a maintainer probably has something to do with being a decision maker for the project, but even this is subjectively defined within each project. A project with multiple working groups might consider those who write documentation and triage issues to be maintainers, because they “own” a certain area of work. Other projects reserve the term for core developers or those with commit privileges, regardless of their influence. And, as with contributors, some developers don’t want the responsibilities of being a maintainer, even if they effectively act like one.

To keep things simple, I'm going to defer to the common definitions of these terms in open source, which are centered around a project's code repository. While I encourage you to consider how these terms could be different—for example, “casual contributors” might be better classified as “users,” and “contributors” could be better reframed as “members”—coming up with new names for these roles is beyond the scope of this book.

- **MAINTAINERS** are those who are responsible for the future of a project's repository (or repositories), whose decisions affect the project laterally. Maintainers can be thought of as “trustees” or stewards of the code.
- **CONTRIBUTORS** are those who make contributions to a project's repository, ranging from casual to significant, but who aren't responsible for its overall success.
- **USERS** are those whose primary relationship to a project's repository is to consume (or “appropriate”) its code.[‡]

Finally, the term “developer” can be confusing when discussing who's doing what in open source. Everybody who interacts with a software project likely identifies *personally* as a developer, which is distinct from the role they play in the project.

You don't have to be a developer to participate in an open source project, and many projects make this point explicit in order to appear welcoming to all contributors. In practice, it's exceedingly rare to find a contributor who's never touched code in their life and who is also excited about participating in an open source software project. (It's hard enough to get experienced developers to feel comfortable enough to participate!) Most contributors start off as users, and the users of open source projects are developers. Even

those who primarily contribute in the form of documentation, design, educational materials, or community events are likely also developers.

In this book, I'll assume that maintainers, contributors, and users are all developers, meaning people who've written and used code for some purpose in their own lives. I'll only use the term "developer" in the general sense, as an identity that's shared among all participants.

THE ROLE OF A MAINTAINER

"Creation is an intrinsic motivator, maintenance usually requires extrinsic motivation."

—@BALUPTON, via [isaacs/github](#)¹³⁹

In his 1975 book *The Mythical Man-Month*, Fred Brooks tackles the problem of organizational design for teams building software. He cites an idea from computer scientist Harlan Mills, who suggests organizing developers like a surgical team, "rather than a hog-butcherling team."¹⁴⁰

In the surgical-team model, there is a "chief programmer," who, like a surgeon, sets the project's specifications and design. The surgeon has a "copilot," who serves as their confidante and right arm. Then there are a number of supporting roles, including someone who handles money and administration, someone who writes documentation, and so on.

But Brooks first proposed this concept at a time when writing code was significantly more cumbersome than it is today. In addition, it can be difficult—politically and practically—to identify and appoint the right "chief programmer" on modern software teams, although informally there

are junior and senior developers, as well as engineering managers who set priorities for the team. It's not common to hear developers employed at your typical software company talk about themselves like a surgical team.

However, the surgical-team concept *is* useful for explaining the different roles in an open source software project, because maintainers have a set of indisputable responsibilities and permissions that other contributors do not. In particular, projects with centralized communities are centered around one or a few maintainers who play the role of surgeon, with other contributors and users supporting their work.

To borrow again from conservation biology, maintainers can be thought of as a *keystone species*. A keystone species is small in population size but has an outsize impact on its ecosystem. If we were to imagine a forest, for example, while there may not be many wolves in absolute numbers, if the wolves disappeared there would be cascading effects on the rest of the forest. The deer population, lacking a natural predator, would grow out of control, the plants would start to disappear because there would be too many deer, and so on.

Similarly, although maintainers are few in number, their impact on an open source project is far-reaching, because they're the bottleneck to everyone else's contributions. Unlike other contributors, maintainers tend to work laterally. When making decisions, they must keep the whole project in mind, balancing competing needs among users and contributors.

Because the term “maintainer” doesn't have a universal definition, it can be difficult to make generalizations about what the role entails. Bigger projects might have maintainers who are responsible for specific subprojects, or for areas of expertise that span the whole project. For example, the Python-

based web application framework Django has a security team, an ops team to maintain tooling and infrastructure, a release team to build and manage releases, and several technical teams to handle triage, contributions, and code review.^{[141](#)} All of these developers could be considered maintainers, but because Django is such a big project their work is more specialized.

In smaller projects, maintainers are more likely to be handling more aspects of the project themselves. They’re the ones who respond to new issues and keep them organized; answer user support questions; maintain tests, style guides, and continuous integration; and write documentation. For example, Caddy, an open source web server, is mostly maintained by its author, Matt Holt, who reviews pull requests, responds to issues, and is the primary core developer.^{[142](#)}

Sometimes, the term “maintainer” is used interchangeably with “core developer,” which raises another question: Do maintainers write new code, or simply tend to the existing code? The term “maintenance” seems to imply reactive work, whereas “core development” implies proactive work. Maintaining a minimum degree of functionality might include tasks like responding to issues, reviewing pull requests, and upgrading software dependencies. But actively evolving the project would require tasks like defining the vision of the project, writing new features, and reducing “scope creep” (not letting in too many features, which eventually bloat the purpose of the project).

We can return to Spencer Heath MacCallum’s idea of “proprietary communities” to understand what a maintainer does differently from other contributors. MacCallum suggests that a proprietor serves three functions: *selection of members, land planning* (in the case of open source, “land” is

the codebase), and *leadership*. Member selection and land planning occur whenever maintainers review pull requests from new contributors. They must choose contributors “with a view to their compatibility and complementarity with other members,”¹⁴³ and they accept contributions based on how they affect the rest of the code.

Maintainers must also weigh the value of the contribution against the cost of maintaining it. A proposed contribution might seem like a good idea on its own, but it’s the maintainer who must tend it, long after the contributor is gone. If the contribution is expensive to maintain, or doesn’t fit the overall vision of the project, they might decide the cost is not worth it.

Maintainers decide who and what goes into a project, not just because they have the ability to merge pull requests but also because someone needs to be mindful of how these contributions fit together, so the project doesn’t end up looking like some strange neural network painting. In this sense, maintenance can be thought of as a form of curation.

Only maintainers are interested in “the success of the whole community rather than that of any special interest within it.”¹⁴⁴ While maintainers may do other work for the project, all other work can be performed by other community members, whereas a proprietor’s tasks are reserved for maintainers alone.

Although being a “maintainer” doesn’t sound very glamorous, it’s hard to come up with a more precise term without sacrificing accuracy. Some projects aren’t actively developed anymore, but they are being maintained. And placing too much emphasis on the proactive work of a project, like writing new code, might undervalue the very real maintenance work that’s needed to support it. Which term to use might just depend on what a

particular project's maintainers, or core developers, are trying to convey about their relationship to it.

The term *author* refers specifically to the developer or developers who were responsible for the project's original release. Not all authors become maintainers or even active contributors. For example, while Sebastian McKenzie, also known as kittens, authored Babel, he is no longer a maintainer of the project.¹⁴⁵

Conversely, not all maintainers are necessarily authors, and who is considered a maintainer may change over the life of the project. Some of these transitions occur after the contributor community has reached a certain size. Jacob Kaplan-Moss and Adrian Holovaty, who authored Django, retired as BDFLs (“Benevolent Dictators for Life”) after nine years, with Jacob explaining that “the longer I observe the Django community, the more I realize that our community doesn't need [us].”¹⁴⁶

Maintainer transitions occur even in smaller projects. Urllib3, a Python HTTP library, has announced multiple “lead maintainer” transitions over its ten years of development.¹⁴⁷ Its author, Andrey Petrov, explains that “handing over the reins of a project is a natural part of a successful project.”¹⁴⁸

If they are not the original author, a maintainer may not have commit or administrative rights, which can lead to exactly the problems one might expect. If an admin disappears unexpectedly, the remaining contributors could be left without a way of merging new changes into the project.

After Jim Weirich, a prominent open source developer in the Ruby community, passed away, the community realized he had not named a

successor to many of his projects. When another developer, Justin Searls, wanted to step in to maintain one of those projects, Rspec-Given, he wasn't able to gain administrative access. Instead, he had to fork the project and convince RubyGems, the package manager for Ruby, to point to the new version.^{[149](#)}

The distinction between "author" and "maintainer" also underscores the inherent tension between creating and maintaining software. Some developers love to make things, but they don't like maintaining them. Functionally, the work required by these two roles is quite different.

If a project is lucky, it'll attract both creators and curators, who can serve complementary roles as co-maintainers. Developers tend to be more excited to create than maintain, so finding someone who enjoys the latter can be a real boon. One maintainer explained to me that the project's author, who's still actively involved, is an "inventor" who loves creating but has little interest in maintaining. By contrast, the person I spoke to loves organizing, curating, and rigorously applying process. They work well together, although sometimes he wonders what will happen when the author eventually steps down. He worries he won't be able to fill the shoes of an inventor. Likewise, if he were to step away from the project, finding someone with his level of patience and attention to detail would be difficult for his co-maintainer.

Sean Larkin, the webpack maintainer, found that his skills doing outreach and user support complemented those of webpack's author and lead developer, Tobias Koppers:

When I first became a maintainer, I was too terrified to make code changes. I didn't know how it worked. So I asked myself what I could

do that I could kick ass at. I essentially just got on Twitter and started spending hours a day searching “Webpack.”¹⁵⁰

Yochai Benkler suggests that all members in a peer production model are intrinsically motivated to participate. While intrinsic motivation explains why developers *create* software, as well as why they might casually participate, it doesn’t fully explain why developers continue to *Maintain* projects over time. And while maintainers’ behavioral incentives are poorly understood, they are extremely important to get right, because they are the linchpin to all other contributions.

Developers initially author a project because it’s fun for them, or they have some problem they want to learn to solve. They’re writing code and working on the things they want to work on. Regardless of whether a maintainer is the author of the project, there’s likely some ongoing creative work required over the life of the project, in the form of writing new code or learning new skills. This sort of work tends to be most intrinsically motivated.

Authors may also accrue extrinsic, social rewards in the form of recognition and reputation. But as the project matures, reputational benefits flatten out: after all, the authors are already known for their affiliation to the project, and spending more time on its maintenance isn’t going to change that. At the same time, reactive work may start to overtake the proactive tasks, while providing neither intrinsic nor extrinsic benefits. This phase of the project is like reading through an ever-growing comments section: maintainers experience diminishing returns on the marginal value of reviewing additional contributions. Eventually, the value derived from new contributors might not exceed the cost of sifting through contributions. It’s

work that maintainers don't really want to do, because there is no clear benefit to doing it.

Rarely is long-term maintenance primarily about sharpening one's skills as a developer. If anything, maintainers sometimes stick with a project *despite* not learning from it or using the technology anymore. They might cite feelings of obligation, community, or helping others as reasons for looking after the project.

It's around this point that maintainers fan out into a few different outcomes. They might discover renewed intrinsic joy in maintaining the project because they're learning a different set of valuable skills, such as project management or leadership. If they don't enjoy these things, however, they must either figure out how to distribute the work to contributors and users or reduce the overall time they spend on undesirable tasks: for example, by turning off issues or auto-closing pull requests. If both avenues prove to be difficult, maintainers will eventually step down, find a replacement, or disappear altogether.

After a maintainer leaves, another developer might step up. A lesser-known developer may be eager for an opportunity to maintain a popular project. Christopher Hiller became the maintainer of Mocha, a JavaScript test framework, after its author, TJ Holowaychuk, stepped down. As he explained in an interview,

[TJ] put a call out that said, "Hey, I need to have somebody take over my projects. They're up for grabs." I at the time was a user of Mocha and said, "Hey, I would like to help. I enjoy using this software and I don't wanna see it die." Basically, he just gave me the commit bit and didn't say anything, and that was it.[151](#)

On the other hand, taming a fire hose of support issues for a project they didn't create, and aren't deeply familiar with, is not necessarily fun for most developers, and the costs may outweigh the reputational benefits. The original author, like a miner who's delved into an ore deposit until it's depleted, may have reaped all the associated status gains from creating the project, leaving nothing behind for the next maintainer. There are many more orphaned projects than there are developers clamoring to become maintainers.

Much like a founder or CEO, a maintainer can't step away as easily as other contributors. In projects with decentralized communities, when individual motivation declines members might leave, but they are replaced by new, eager faces. For solo maintainers, however, when their motivation declines the decision to leave can have serious consequences for the project.

Dominic Tarr is a developer who's created hundreds of popular npm modules, many of which are widely downloaded and relied upon by millions of people around the world. In November 2018, he gave commit rights to a stranger who claimed they wanted to help maintain one of these modules, event-stream, which Tarr had long moved on from. Instead, the stranger proceeded to insert malicious code into the module, in an attempt to steal money from users of an application that depended on the library.

As hundreds of developers scrambled to figure out what was wrong and fix it, Ayrton Sparling, the developer who discovered the hack, made a frightening comment that, while the code appeared to be malicious, "the worst part is I still don't even know what this does . . ." [152](#)

Many developers were angry, calling Tarr's actions irresponsible and careless. But Tarr stood by his decision. He published a statement on

GitHub, describing why maintainers hand off access to others:

I didn't create this code for altruistic motivations, I created it *for fun*. I was learning, and learning is fun. . . . I've written way better modules than this, the internet just hasn't fully caught up. . . .

If it's not fun anymore, you get literally nothing from maintaining a popular package.

One time, I was working as a dishwasher in a restaurant [*sic*], and I made the mistake of being too competent, and I got promoted to cook. This was only a 50 cents an hour pay rise [*sic*], but massively more responsibility. It didn't really feel worth it. Writing a popular module like this is like that times a million, and the pay rise [*sic*] is zero.¹⁵³

Tarr explained that, far from recklessness or a mistake, handing off maintenance to strangers was considered a best practice among many JavaScript developers. He referenced a popular blog post by developer Felix Geisendörfer, who explicitly endorses the strategy of defaulting in order to giving new contributors commit access:

Somebody sent a pull request for a project I was no longer using myself, and I could see an issue with it right away. However, since I no longer cared about the project, and the person sending the pull request did, I simply added him as a collaborator and said something like this: "I don't have time to maintain this project anymore, so I gave you commit access to make any changes you'd like."¹⁵⁴

We're used to thinking of publication as signifying the end of responsibility, as when a writer publishes a book, or a pianist finishes a performance. An

open source maintainer, on the other hand, is expected to maintain the code they published for as long as people use it. In some cases, this could be literally decades, unless the maintainer formally steps away from the project.

This is the equivalent of a writer being asked to edit and make changes to the same book every day, into perpetuity, long after they've reaped the initial financial and reputational rewards from its creation. What's more, unlike other content, open source code is relied upon by people, companies, and other institutions that need it to keep working, long after the maintainer's interest has waned. External contributions don't necessarily reduce the burden of maintenance either, because they still require someone to review and merge them.

ACTIVE AND CASUAL CONTRIBUTORS

The contributors to an open source project can be classified as either *active* or *casual*, which is often determined based on the frequency of their contributions.

There is no clear line that divides these two categories, however: what constitutes a “frequent” contributor varies depending on the project. Researcher Na Sun et al. argue that we should not rely on objective criteria (like number of contributions) to define these groups, but should rather define them by activity relative to other members *within* their community:

The size, topic and culture of an online community may influence lurking behaviors. For instance, small online communities that focus on technical topics usually have fewer members but a higher participation rate than large online communities that cover various topics. Thus, the lurkers in technical communities may be considered posters in [other] communities.^{[155](#)}

Rather than considering frequency, then, it’s better to look at contributor motivations to understand how these two groups differ.

Active contributors (also called “regular contributors” or “long-term contributors”) are considered members of the project, based on their reputation or the consistency of their contributions. They’re what we typically imagine when we think of open source contributors: a community of developers in which members are invested in one another and in the project.

Not every project has active contributors, depending on its size. Whereas a larger project might have three types of contributors—*maintainer*, *active*, and *casual*—a smaller project might just have a few *maintainers* and many more *casual contributors*.

Kraut and Resnick note that people are more willing to contribute when the group is small and “they think their contributions make a difference to the group’s performance.”¹⁵⁶ Following the collective effort model they cite, active contributors aren’t necessarily a function of how many users the project actually has, but rather of how many perceive that they can make a meaningful contribution. If the project doesn’t require much work, or if a single maintainer retains a lot of specialized knowledge, it won’t attract as many active contributors.

Active contributors tend to be somewhat specialized, whether they focus on a specific part of the codebase or contribute their expertise, like security knowledge or managing the community. They’re distinct from maintainers in that they aren’t responsible for the overall direction of the project. While active contributors might care about the community, they’re not expected to make trade-offs between their interests and others’.

Because they’re more familiar with the project than casual contributors are, and are better known to the maintainers, active contributors tend to have higher-quality contributions, and they’re more likely to get reviewed and merged by maintainers.¹⁵⁷ A study by Suvodeep Majumder et al. found that 85% of the open source projects studied had “heroes,” meaning developers who participate in at least 80% of discussions related to a commit, and that these developers’ commits contained significantly fewer bugs than others’.¹⁵⁸

While contributors are sometimes described as moving through a “funnel” from user —> casual —> active —> maintainer, there’s evidence that incentives to participate differ between casual and active contributors from day one.

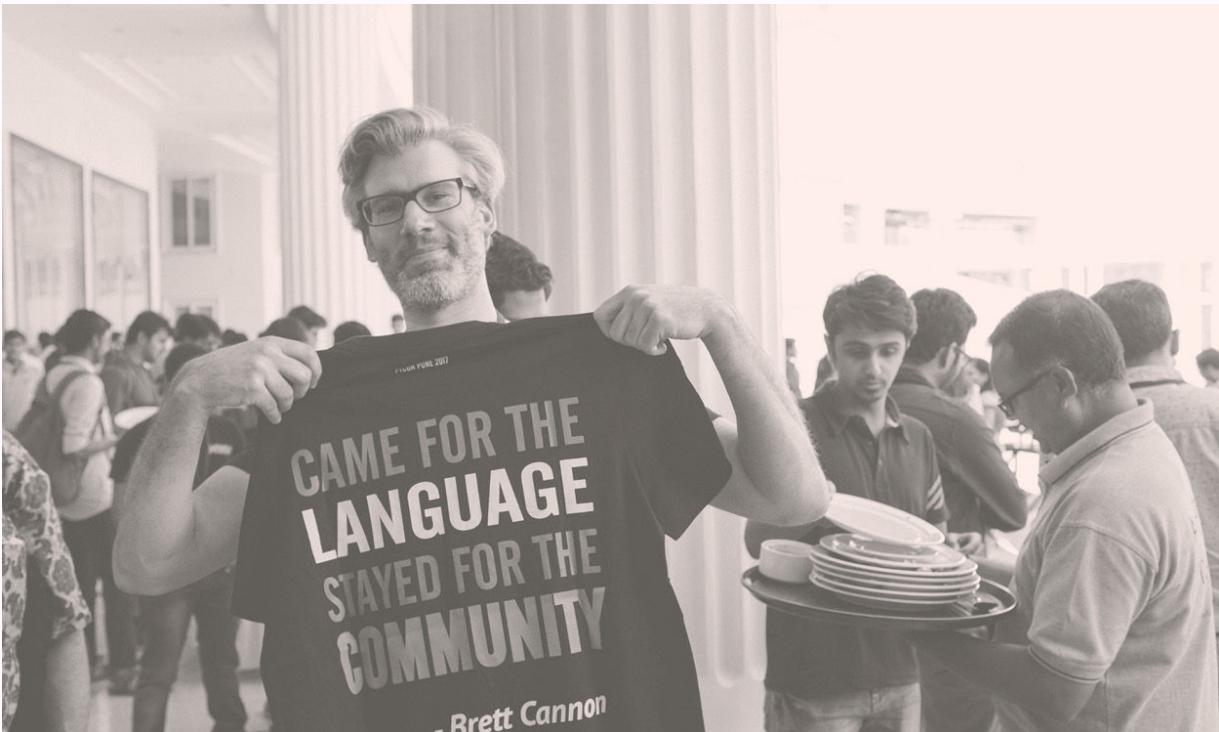
Active contributors might look like casual contributors at first, in that both are “first-time contributors,” but their motives are different. During their first month, active contributors tend to exhibit prosocial attitudes, demonstrating higher levels of motivation to help others.

Whereas a casual contributor might begin by opening an issue or pull request (“I need something”), for example, an active contributor might start out by commenting on someone else’s issue (“I want to help with something”).¹⁵⁹ Another common behavior pattern of active contributors is lurking on mailing lists, issue threads, and pull requests before interacting with the project.

Minghui Zhou and Audris Mockus, in their study of long-term contributors on Mozilla and GNOME, found that “joiners who start by commenting on instead of reporting an issue or ones who succeed to [sic] get at least one reported issue to be fixed, more than double their odds of becoming a long-term contributor.”¹⁶⁰

Active contributors who stick around do so not because they successfully made their first contribution, but because they already came in with the intent to stick around. In other words, they were never really casual contributors in the first place. (Of course, a project that makes it difficult for newcomers to contribute will discourage would-be active contributors, as well as the inverse.)

An active contributor is often motivated by a desire for community, but they might have secondary motivations around reputation and learning. They enjoy the social aspects of the project and want to feel as if they’re “part of something.” Brett Cannon, the Python maintainer, summarizes this mindset as such: “I came for the language, but I stayed for the community.”^{[161](#)}



An attendee holding up a conference T-shirt for PyCon Pune 2017.

Photograph courtesy of Kushal Das.

Active contributors have lower churn than casual contributors because they see themselves as part of the project. They’re more likely to leave due to a change in life circumstances than pure boredom or disinterest. As Kazuhiro Yamashita et al. describe it, “Like citizens who become accustomed to their environment, developers who contribute to a project are likely to continue contributing to the same project.”^{[162](#)}

By contrast, *casual contributors* (sometimes called “drive-by contributors”) have a transactional relationship with the project.

Casual contributors are, generally speaking, those who have made one contribution or less (such as an unmerged pull request) to the project. However, it’s easier to identify them by their low affinity to the project. A casual contributor may have made multiple contributions, but this type of contributor’s defining characteristic is that they don’t feel a deeper connection to the contributor community, beyond a desire to see their own contributions merged. Viewed through the lens of membership, they are better understood as users than contributors.

Note that the definition of a project’s “contributor community” varies widely, as we’ve seen in previous examples. A developer might still feel affinity to a project, despite contributing infrequently, if they’re involved with a related project. (Recall Sean Larkin’s comments about Lerna, despite identifying as a maintainer of a different project, webpack.) We might assume this person is a “casual contributor” based on the frequency of their contributions, but in reality they function like an active contributor (e.g., higher-quality contributions, higher chance of pull requests being merged).

Generally speaking, truly casual contributors are current users who have discovered an issue that needs addressing: most commonly fixing a typo or a bug, but sometimes also adding new features or *refactoring* code (rewriting code to reduce its complexity, like editing one’s writing).¹⁶³ Rather than participating in a discussion, their first action on the project is probably opening an issue or pull request. Whereas active contributors show interest in adding value to others early on, casual contributors demonstrate an acute, personal need at the outset.

Casual contributors are in an “I want this fixed, and then I’m done” mindset. In an online community, the equivalent of casual contributors would be a casual commenter: someone who participates in discussions or asks questions that satisfy their own interest, rather than for the benefit of others.

Casual contributors don’t plan on sticking around, and they have the least amount of context regarding the project’s process or norms. They only want to know enough to get their contribution merged. Their primary interest is personal, and it’s the maintainer’s job to figure out how to weigh the contributor’s needs against those of the project.

It’s tempting to attach a host of assumptions to casual contributors—that they are unfamiliar with open source, new to the project, demanding, destructive—but it’s better to simply think of them as self-oriented rather than communally oriented. A casual contributor seeks a resolution to their own problem; that their contribution might benefit others is merely a positive externality.

Casual contributors are like tourists visiting New York City for a weekend. Just as we wouldn’t expect, or even want, tourists to participate in local governance decisions, we shouldn’t assume that casual contributors are part of a project’s contributor community simply because they are physically present. After studying why newcomers abandon open source projects, researcher Igor Steinmacher et al. conclude, “By reading the discussions initiated by dropouts, we realized that part of the newcomers had no intention to join the project.”¹⁶⁴

Where exactly to draw the boundaries around a project’s “community,” however, is, once again, more art than science. A project’s community

might extend beyond a specific repository, nested within a broader ecosystem, similarly to how a town has its own local government but also falls under the jurisdiction of a state or provincial government, as well as that of the national government. Does the “community” of an npm package include that specific package, npm, or JavaScript as a whole? While there is no consistent answer across projects, we might be able to spot these affinities based on how developers self-identify. There are very few developers who would call themselves part of the “Babel community,” since Babel is such a specific package, but many more who might contribute to Babel, or care about its future, based on its position in the React community.

In the context of casual contributors, then, what constitutes “casual” is relative to the community in question. Dan Abramov is a well-known React developer, but he doesn’t contribute regularly to Babel. In the context of Babel’s governance, he might be considered a casual contributor, but because he’s an active contributor to the React ecosystem his contributions to Babel would be more trustworthy than those of an unknown developer.

If maintainers are the nucleus of a project, casual contributors are the electron cloud, darting about the periphery. They represent the inverse of the maintainers’ linchpin status: many in number, but little aggregate value.

The ratio of casual to active contributors varies greatly between projects, depending on the size of their contributor community and how these terms are defined. One study suggests that casual contributors make up three-quarters of all contributions.¹⁶⁵ Pandas, a Python library for data analysis, lists over 1,400 contributors, but just four developers contributed nearly half of all commits in 2018.¹⁶⁶

Casual contributors have always existed in open source. Kraut and Resnick, citing a study from 2005, report that “fifty-four percent of developers who registered to participate in the Perl open-source development project never returned after posting a single message.”¹⁶⁷ They note that this behavior is not unusual among online communities more broadly. For example, “68 percent of newcomers to Usenet groups are never seen after their first post,” “sixty percent of registered editors in Wikipedia never make another edit after their first twenty-four hours participating,” and “forty-six percent of the members of guilds in World of Warcraft leave their group within one month, generally migrating to other groups rather than abandoning the game itself.”

However, the extent to which casual contributors affect projects today is a newer phenomenon, a direct byproduct of GitHub’s platform effects. By lowering the friction to contribute, combined with the exponential growth in *users* of open source projects, GitHub has created the ideal conditions for casual contributors to flourish. For projects with few maintainers that lack an active contributor community to moderate these effects, casual contributors can be overwhelming, like too many tourists flooding into a town.

Because they’re less familiar with the project, casual contributors have different needs than active contributors. Casual contributors are more likely to struggle with both the social and technical aspects of contributing: setting up their environment, understanding the codebase, and meeting the contribution requirements.¹⁶⁸ As a result, their contributions tend to be lower quality and take longer to resolve, and are generally less likely to be merged.¹⁶⁹ It’s therefore important for maintainers to provide casual contributors with the documentation and tooling they need to write and

submit high-quality contributions on their own, which reduces headaches on both sides.

The combination of unpredictable quality and high volume means that casual contributions can quickly turn into a time sink for maintainers if they’re not careful. Much like a YouTube creator must learn to manage their relationship with the audience in order to retain their desire to keep making videos, a maintainer must manage their relationship with casual contributors in order to retain enthusiasm for the project.

For maintainers, it’s helpful to think of contributors as falling into these two distinct “tracks,” identifiable based on self-interested versus prosocial motives, so the maintainers can decide how much time they want to spend on a particular contributor, as well as how to tailor the resources and support that they offer.

Not every maintainer has the same policies for handling different types of contributors. Some maintainers enjoy spending time helping every contributor make their first contribution, regardless of how long they stick around. Other maintainers are more protective of their attention, so they limit their focus to contributors who’ve demonstrated longer-term interest. Regardless of the position that a maintainer adopts, it’s clear that not every contributor participates in a project for the same reason.

ACTIVE AND PASSIVE USERS

Finally, an open source project has users, who might be thought of as the “silent majority.” They use the project’s code, but they don’t self-identify as contributors.[§]

Generally speaking, maintainers don't know who their users are. Like lurkers in an online community, users can consume code without necessarily making themselves publicly known. Maintainers will have only a vague sense that their project is widely used based on the growing volume of issues or pull requests, or when users ask questions or reach out to them. In 2019, GitHub added a “dependents graph” to repositories for certain programming languages, which shows the packages and projects that depend on it.

A maintainer may only discover who depends on their project when a user steps out from the shadows: for example, by opening an issue or pull request, or by reaching out to a maintainer directly. Daniel Stenberg, author of the command-line tool cURL, didn't realize he had created one of the most widely used software projects in the world until people told him “they saw his name in the ‘about’ window of software, or buried in documentation,” many years into cURL’s development.^{[170](#)}

Sometimes, these moments of discovery are heartwarming. When a SpaceX engineer tweeted at Jeff Forcier, who maintains a Python library called Fabric, “You do know we use the hell out of fabric, right?,” Forcier tweeted out a response: “TIL that SpaceX uses my software! The feels are real.”^{[171](#)} 

The use of tracking and analytics is viewed unfavorably by a vocal number of developers (though it's not clear that they represent a majority), who cite privacy reasons. Perhaps the bigger issue is that not all open source projects can track users anyway, from a technical perspective. Of the projects that could track usage, most maintainers don't bother with it, either due to lack

of interest or difficulty of implementation, or perhaps to avoid public controversy.

Some projects brave it anyway. The team behind Homebrew, a macOS package manager, uses analytics to help maintainers make better decisions about the project's development needs, explaining that “anonymous aggregate user analytics allow us to prioritise fixes and features based on how, where and when people use Homebrew.”¹⁷² Their decision to add tracking, announced in 2016, received mixed reviews among users. One commenter on Hacker News complained, “Why are you even collecting this information? Homebrew isn't some for-profit product trying to optimize its funnel. Keep doing what you were doing. You were doing a great job. There is no benefit to you or us to silently spy on us.”¹⁷³ Nonetheless, most users don't opt out of Homebrew's tracking.

Feross Aboukhadijeh, who maintains a style guide and error-catching tool called Standard JS, used an opt-in strategy to figure out who his users are. He created a public space for users to identify themselves by opening an issue on his project called “Which companies are using ‘standard’?”¹⁷⁴

Given that users aren't usually visible to maintainers, we can characterize them as default-passive: They just want something that works. If users don't make themselves known, it implies they are happy enough with the direction of the project that they don't seek further interaction with its developers.

It's also often the case that developers use a project without even knowing that it exists. A dependency file could contain hundreds of packages, all of which are installed when a developer runs a single command like “npm

install” or “bundle install.” And open source projects like OpenSSL and Expat live below the application layer, yet are still widely depended upon.

Although maintainers don’t always have a direct relationship to their users, some users *do* make contributions to open source projects, even if they don’t think of themselves as contributors. Common examples include:[¶]

- **EDUCATION**, in the form of live-coding, or making videos, blog posts, or tutorials about the project
- **SPREADING THE WORD**, in the form of organizing events, posting about the project online, or giving talks at meetups or conferences
- **SUPPORT**, in the form of answering user questions on mailing lists, issue trackers, or chat
- **BUG REPORTS**, in the form of filing an issue with the project after discovering a bug^{**}

These *active users* are similar to active contributors, but they tend to operate independently from the project’s contributor community. It’s possible they’ve never interacted with the project’s repository at all.

Active users can be thought of as “satellite” communities, which is why they are treated separately here. While an active user is a type of contributor, they may not self-identify as a contributor or even as part of the project’s community, preferring to think of themself as “just a big fan” of the project, or as motivated by wanting to demonstrate their own expertise.

Active users exhibit traits of both active and casual contributors. Much like casual contributors, active users don’t necessarily feel an affinity to the project’s contributor community. They’re motivated to make tutorials, organize events, report bugs, or give feedback on new releases, but they

mainly do so with their own interests in mind, whether reputational benefits or the satisfaction of solving their own problems.

Like active contributors, however, active users tend to think on a longer time span because of the reputational benefits associated with building and demonstrating expertise, whether directly (e.g., as a top answerer on Stack Overflow, or by becoming a recognized expert or evangelist) or indirectly (e.g., self-taught expertise might lead them to new opportunities, like a new job).

Active users can be a maintainer’s best ally or worst enemy. At their best, active users test new releases, report bugs, and shield maintainers from, as researchers K. Crowston and J. Howison put it, “a barrage of setup, configuration, and build questions from passive users—those who use the code without contributing themselves.”¹⁷⁶ They form the backbone of a user-to-user support system that can reduce a maintainer’s workload. Similarly to active contributors, active users tend to have deeper insight into the project than do casual contributors, and their names are often recognized by maintainers, even if maintainers don’t work with them directly. At their worst, however, active users can be pushy, putting stress on the project by demanding work, or providing other users with “unofficial” answers that are outdated or inaccurate.

ASSESSING THE HEALTH OF A PROJECT

For clubs and federations, we can get a sense of how well a project is doing based on the size and growth of its active contributor base. But for stadiums, most contributors are casual. Contributor count doesn't tell us whether these projects are doing well or poorly; if anything, too many casual contributors could signal that a project's maintainers are overwhelmed.

What does it mean for a project to be doing “well”? There are three success metrics that are interrelated but distinct:

- **POPULAR** refers to how many people use the project. For example, a testing tool or text editor could be popular, even if other software doesn't directly depend on it.
- **DEPENDED UPON** refers to how much software *depends* on the project, meaning that it is being actively used by other software, whether directly or transitively. If the project were to disappear, it would break someone else's software.
- **ACTIVE** means the project is actively developed; this communicates an expectation that the project will continue to be maintained in the future. If a developer were evaluating whether to use this project, how would they know it's reliable?¹⁷⁷

A project could be popular, meaning that it is widely used. It could be widely depended upon, meaning that a lot of other software relies upon it. But a popular and widely depended-upon project might still not be actively developed.

One maintainer, who has created hundreds of popular libraries, described to me what he does when he gets tired of maintaining them. Using a “ghost” admin account that he has purposely forgotten the password to, he transfers his unwanted projects to that account, then removes his main account as administrator. This way, he explained rather cheerfully, nobody will bother him about maintenance, because he cannot access the project anymore, even if he tries. If anybody wants to change the code, instead of bugging him they’ll simply have to fork and maintain it themselves. While this practice works well for him, it illustrates how a popular and widely depended-upon project could have literally nobody at the helm.

GitHub offers a less extreme version of this practice with its “archive” feature, as well as the ability to turn off issues (but still accept pull requests). When a developer archives their project, anyone can still download and fork the project, but issues and pull requests are turned off. Even the project’s owner cannot make changes to the project unless they un-archive it first. The archive feature helps developers publicly communicate that a project is no longer actively maintained, and that users should download and use the code at their own risk. (Unofficially, maintainers also sometimes write “UNMAINTAINED” in the project’s description or README.)

Measuring the health of a project requires measuring the fitness of its developers. The *truck factor*, or *bus factor*, mentioned briefly at the beginning of this book, is perhaps the simplest and best-known heuristic.

A project’s bus factor is the number of contributors that would need to get hit by a bus before the project is compromised. For example, if a project has a bus factor of 1, that means there is only one maintainer, who, if they were

hit by a bus, would take all their knowledge of the project to their grave. The implication is that projects with higher bus factors are more resilient, because knowledge is distributed among more people.

Contributors are commonly used as a measure of project health, whether total contributor count or contributor growth rate. GitHub prominently displays total contributor count on a repository's homepage, and this number is frequently cited, with pride, as a sign of a healthy, booming project.

Contributor count is a useful metric for clubs especially. If we were to compare an open source project to a meetup group, it seems sensible that more members would indicate that the group is thriving, while a dwindling membership might suggest that things aren't going so well.

At the extreme, there is some value in looking at contributor count, regardless of project type. If a project had zero contributors, all work would cease. Like any ecosystem, *speciation* (creation of new species, or, in this case, new contributors) must outpace *extinction* (death of existing species, or, in this case, loss of contributors) in order to survive.^{[178](#)}

For stadiums, however, contributor count is misleading, because it obfuscates who's doing the most work. By only looking at total count, contributors are assumed to be *, or interchangeable, meaning that any one contributor should theoretically be able to step in and perform the work of another.*

The idea that contributors are fungible is rooted in the theory of the commons. If one member's interest wanes and they fade away, someone else will appear to take their place. So long as there are enough new

members coming in, and enough members who stick around, this roundabout can go on indefinitely.

But as Fred Brooks notes in *The Mythical Man-Month*, “Men and months are interchangeable commodities only when a task can be partitioned among many workers with no communication among them”—in other words, the idealized version of the commons.¹⁷⁹ In practice, contributors are not all the same. Some are more productive, while others stick around longer or offer certain specialized skills. These differences are a function of skill level, familiarity with the project, and motivation. A project’s maintainers need to think differently about how they attract new contributors who are willing to triage issues, versus contributors who want to write new features.

“Contributor count” is not sufficiently nuanced to distinguish *how* an individual contributes, and this is especially visible among stadiums, where a few maintainers do most of the work. Removing a maintainer from an open source project causes significantly more harm than removing a one-time casual contributor.

Two projects could each have a hundred contributors, but if one has a hundred active contributors, and the other has two maintainers and ninety-eight casual contributors, these tell two very different stories about the project’s health. In a blog post, Ruby developer Richard Schneeman highlights the limitations of these metrics by comparing a Ruby library called Sprockets, which lost its primary maintainer, to Rails itself:

From 2011 to 2016 Sprockets has had 51 million downloads, and I’d like to put that into perspective. Rails has had 65 million downloads, so Sprockets is pretty close, and, of that entire library, one developer is

responsible for 2027 commits, which happens to be about 68% of Sprockets. That's one person. Compared, in contrast, to a Ruby hero, Rafael Franca, who has over 5000 commits on Rails. This accounts for only 0.9% of Rails.^{[180](#)}

We need to look beyond the total number of contributors, and instead consider contributor quality. Rather than measuring the quality of *code* that a contributor writes—which would be difficult to objectively evaluate—we could use the quality of a contributor's *reputation* as a proxy. For example, if a contributor is frequently mentioned on other issues, pull requests, and code reviews, this suggests that other developers perceive they are important to that project. If their pull requests are reviewed more quickly than others', this suggests that they are more trusted.

But measuring contributor quality matters less on projects where just a few maintainers do most of the work, and it's clear to everyone who those people are. We still need a health metric that translates across different project types. One option is to focus on measuring *work done*, meaning a project's overall level of activity, rather than measuring its contributors.

When a developer is trying to decide whether to use an open source project, one of the first things they'll glance at is the date of the project's last commit. A project that hasn't seen any commits in several years is probably less maintained than one whose latest commit was a few days ago.

“Work done” is easier to generalize than contributor count is, allowing us to compare across project and contributor types, since a project's overall level of activity is agnostic to how the work gets done.

We can form a picture of a project’s overall activity by first looking at the volume of work required, such as *commits* (both the date of the latest commit and the rate of new commits) and the *rate at which issues and pull requests are opened*. As discussed at the end of Chapter 2, a project might not be actively developed, but it can still be considered healthy if there isn’t much work required. Developer activity off of GitHub (such as on Stack Overflow, chats, and mailing lists) can help paint a more complete picture of how much work is involved.

Next, we can look at the rate at which that work gets completed. Ideally, new interactions (such as opened issues and pull requests) are addressed quickly, which often depends upon maintainer responsiveness. A few metrics related to responsiveness include:

- **NUMBER OF OPEN ISSUES AND PULL REQUESTS:** A high number of open issues or pull requests suggests the project is not well maintained, although projects have different philosophies as to whether they prefer to keep issues open or closed.
- **AVERAGE TIME TO FIRST RESPONSE**, from a maintainer, on a new issue or pull request. (Sometimes, maintainers use bots as first responders.)
- **AVERAGE TIME TO CLOSE AN ISSUE OR PULL REQUEST.**¹⁸¹

Activity that directly feeds back into “work done” (meaning, changes to the code) should weigh more heavily than issues and questions. The latter might help us measure a project’s *popularity*, but a project that’s all talk and no action (lots of discussion, but no commits) is not necessarily healthy. By contrast, “issue zero” and “pull request zero,” with a steady rate of issues

and pull requests opened and closed, would suggest that a project is in comparatively better shape.

No metrics are perfect, however. Data can give us insights, but we still need to write the story. A few important caveats to keep in mind when developing project health metrics:

- **LACK OF GRANULARITY:** A project could have one massive pull request that takes months to get merged, versus a pull request containing a few typo fixes—yet each of these would count as one pull request. Commits are more granular than pull requests, but they aren’t uniform in size either.
- **WORK RATE VARIES ACROSS PROJECTS:** Depending on the work required, some projects might have consistent but low activity, while others projects have inconsistent large bursts of activity. It’s likely that there are multiple, stable pictures of “project health” that could be identified.
- **PROJECTS THAT ARE STABLE:** If a project is stable and widely used but doesn’t require much work, it shouldn’t be penalized for low activity. For this reason, issues are still important to track along with pull requests, even if they’re weighted less. For example, a project that’s not very active based on commits but has many open, unanswered issues might not be in the middle of active development.¹⁸²

Looking at a project’s output, rather than its number of contributors, prompts a new question: What actually goes into those issues and pull requests? What *is* the work that maintainers need to do, beyond the initial authorship of a project?

Software is frequently characterized as “zero marginal cost,” meaning that it can be distributed for nearly nothing, regardless of how many additional people consume it. If writing open source code is intrinsically motivated, as Benkler suggests, and if software did actually scale to mass consumption at zero marginal cost, we would have a very tidy situation indeed.

But the problem is not quite that simple. Code is nearly free to distribute, but maintenance can still be expensive. Although it’s not immediately visible from the code itself, software *does* incur a hidden set of costs over time.

The work of an open source developer goes beyond the initial costs of creation. Maybe developers can’t help but make things, and share the things they make, but every time they do, and every time they find success, a tiny, invisible clock begins to tick, and they’re tasked with managing the care and feeding of their code into perpetuity.

^{*} Common pool resources are like public goods (for example, air), in that it’s hard to exclude anyone from using them. But, unlike public goods, they’re a depletable resource. If I cut down wood in a forest, that means there’s less wood available to you, whereas my breathing air doesn’t significantly affect your ability to breathe more air.

[†] Several developers also pointed out that this change was not valid in any case, as it is unenforceable based on MIT’s licensing terms and would have required relicensing prior contributions.

[‡] In this context, the term “users” refers to those developers who directly use the project to write code, not end-users who consume software written using that code, or those who otherwise transitively

depend on the project. Think React developers, rather than anyone who uses applications that were written in React. In their book *Understanding Knowledge as a Commons*, Elinor Ostrom and Charlotte Hess define users as “those appropriating digital information at any point in time.”¹³⁸

§ Note that usage is distinct from downloads: there are developers who might download—or in GitHub’s nomenclature, “clone”—a project’s code locally, but not actually ever use it.

¶ This list is roughly organized by user contributions that require the biggest contributor community size, in descending order. It’s unlikely you’ll see people organizing conferences for tiny libraries versus programming languages, for example, but pretty much every popular project, regardless of size, has users who report bugs.

** A study of the Mozilla Firefox web browser project found that among the 150,000 issue reporters over eleven years, there was “a comparably small group of about 8,000 experienced, frequent reporters” (roughly 5% of reporters) who had higher-quality insights and contributions.¹⁷⁵

PART 2

HOW PEOPLE MAINTAIN

THE WORK REQUIRED BY SOFTWARE

04

CODE AS ARTIFACT, CODE AS ORGANISM

“One of my hypothesis [sic] is that species of technology, unlike species in biology, do not go extinct. When I really look at supposed extinct species of technology, I find they still survive in some fashion. A close examination of by-gone technologies shows that somewhere on the planet someone is still producing it.”

—KEVIN KELLY, “Immortal Technologies”¹⁸³

To the untrained eye, writing software appears to be all about the new and shiny, free from the earthly troubles of working with atoms rather than imaginary bits. In practice, software ages quietly, in the shadows, and stubbornly refuses to die.

There are two observations to make about software here, which will help illuminate the problem. Firstly, software, once written, is never really finished. It might be feature-complete, but, in order to continue running, software almost always requires some sort of ongoing maintenance. At minimum, that might mean keeping dependencies up-to-date, but it might also mean things like upgrading infrastructure to meet demand, fixing bugs, or updating documentation.

So-called “greenfield” projects—those where a developer gets to write software from scratch—are coveted for a reason. Most of the work that software developers do is not writing new code, but rather tending to the code that someone else has written. Nathan Ensmenger, a professor of informatics and computing at Indiana University, suggests that “most computer programmers begin their careers doing software maintenance, and many never do anything but.”¹⁸⁴

Fergus Henderson, a software engineer at Google, states that “most software at Google gets rewritten every few years.”¹⁸⁵ Software changes over time as its environment—the other technology *around* it—changes. Henderson also points out that regularly rewriting software is inherently beneficial. It helps cut away unnecessary complexity that has accumulated over time, as well as transfer knowledge and a sense of ownership to newer team members.

The cost of maintenance, coupled with a lack of intrinsic motivation to maintain, is why large open source projects tend to become modular as they grow. Rails’ core developers, for example, incorporated many of the extensions written by the community into the main project until Rails 3, when they switched to a modularized approach. It had become too expensive to maintain other developers’ code under one project, and the maintainers risked sacrificing speed and customizability if they didn’t split it up.¹⁸⁶ Rails merged with a competing framework, Merb, which was known for its modular approach, to help the project scale and meet contributor demand.¹⁸⁷

A second observation is that once software finds a set of users, it’s hard for it to ever really disappear. Someone out there is probably going to use that code for a very long time.

Some of the oldest code ever written is still running in production today. Fortran, which was first developed in 1957 at IBM, is still widely used in aerospace, weather forecasting, and other computational industries. COBOL, another programming language, was first released in 1959. Network Time Protocol, used to synchronize time between computer systems, was initially developed in the early 1980s.

In a cheeky post about the difficulties of phasing out programming languages, Byrne Hobart, who writes about finance and technology, declares that “we either need to pay people to learn [COBOL] or put them in jail if they try.”¹⁸⁸ He continues,

COBOL is a notoriously bad language. It locks programmers into a bunch of annoying conventions

What it has going for it is that it was used by some of the earliest companies that adopted computers. In other words, it’s used by banks. . . . [They’re] peculiarly sensitive to technical risk. If the software works, there’s a strong incentive not to change it.

The madness of having to support older technologies, which are often out of a developer’s control, can drive them to desperate decisions. Anyone who has worked in application development knows the difficulties of making software compatible across browser versions and mobile platforms. In a post ominously titled “A Conspiracy to Kill IE6,” developer Chris Zacharias describes the pain of having to support the infamously burdensome Internet Explorer 6 during his time at YouTube:

IE6 had been the bane of our web development team’s existence. At least one to two weeks every major sprint cycle had to be dedicated to fixing new UI that was breaking in IE6. Despite this pain, we were told we had to continue supporting IE6 because our users might be unable to upgrade or might be working at companies that were locked in.¹⁸⁹

Ultimately, Chris’s team secretly dropped support for IE6 altogether, without telling anyone else at the company.

As JavaScript becomes increasingly popular in web development today, these maintenance costs are especially important to consider. From a backward-compatibility perspective, JavaScript is particularly difficult to support, because it can be executed on both the client and the server side.

In a server-side application, the developer decides which version of software they want users to experience, regardless of what the browser wants. In a client-side application, the version is determined by the user's browser. This means that if browsers decide to start only supporting a newer version of JavaScript, the website will break, even if the developer hasn't changed their code.

From a governance perspective, JavaScript is managed somewhat differently from other programming languages, in that it's officially an implementation of ECMAScript. Whereas most programming languages are open source projects, ECMAScript is a specification, created and managed by a standards organization called Ecma International. Any changes to JavaScript are discussed and approved by Ecma's Technical Committee 39, also known as TC39, comprised of companies and organizations that pay for their membership. (This process is not dissimilar to emoji, which are part of the Unicode Standard, managed and maintained by the nonprofit Unicode Consortium.)

TC39 faces a difficult task in deciding which versions of JavaScript to support. If they drop support for an older “word” in JavaScript, they risk breaking websites that haven't been touched in twenty years, and were abandoned by their owners long ago. These websites are like zombies: they're still accessible to users, but their maintainers are nowhere to be found.

In 2018, the so-called “SmooshGate” controversy arose around a particular JavaScript method. (Methods are like verbs, or actions, for code.) In this case, the method was called `Array.prototype.flatten`, and it had been implemented eight years earlier by a popular library called MooTools in a way that conflicted with modern specifications. TC39 had to decide whether to continue supporting an outdated method for the sake of not breaking old websites or to break them for the sake of evolving JavaScript, the language.¹⁹⁰ A heated debate ensued, resulting in a titillating suggestion, made by developer Michael Ficarra, to rename “flatten” to “smoosh” (complete, of course, with an adorable bunny GIF).¹⁹¹

rename flatten to smoosh #56

Closed michaelficarra wants to merge 1 commit into `master` from `smoosh`

Conversation 97 Commits 1 Checks 0 Files changed 1

michaelficarra commented on Mar 6, 2018 Member ...

As reported in [Bugzilla bug 1443630](#), 8+year old versions of MooTools conditionally define an incompatible version of `Array.prototype.flatten`. See [mootools/mootools-core/Source/Core/Core.js](#) for the responsible code. In an attempt to turn a negative situation into a positive one, I am taking this opportunity to rename `flatten` to `smoosh`.



Michael Ficarra’s “smoosh” proposal.

Software doesn't die, because someone out there—someone its developers may not even be aware of—will continue to use it. The author Neal Stephenson once described Unix as “not so much a product as it is a painstakingly compiled oral history of the hacker subculture. It is our Gilgamesh epic . . . Unix is known, loved, and understood by so many hackers that it can be re-created from scratch whenever someone needs it.”¹⁹² Code is not a product to be bought and sold so much as a living form of knowledge.

But not all old code is actively depended upon, even if it continues to exist. For example, the code used to manage Apollo 11’s command and lunar modules, first written in 1969, was published on GitHub in 2016, meaning that it is still publicly accessible today.¹⁹³ We can view, fork, and modify Apollo 11’s code, but it’s widely understood that the code is archival. The code that is available on GitHub was digitized from a hard copy that lives in the MIT Museum; it’s not expected that anyone will run it today.

Apollo 11’s code is roughly the same age as C, a programming language first released in 1972, three years after Apollo 11’s code was written. C still powers much of our underlying software and hardware today, including the Windows, Linux, macOS, iPhone, and Android kernels. Both C’s and Apollo 11’s code were written around the same time, but while we understand that Apollo 11’s code is a historical exercise, we actually do need C to keep working, because so much of our daily life depends on it.

The difference between these two codes highlights an important characteristic of code, which is that it can exist in either *static* or *active* state.

Apollo 11's code lives in static state: It's a commodity, like lumber. When we treat code as a commodity, we're thinking about it on the basis of pure consumption. It can be bought and sold and traded. We can inspect the code for historical purposes, but it's not used in production.

Code is always published to GitHub in static state. A developer could leave it untouched for fifty years, and it would still look exactly the same. Other people can copy and download the code without costing its author anything. They might also fork the project and do something else with it, but in doing so they produce a different commodity entirely.

In this state, code is easy to copy, share, and distribute. But the purpose of consuming code is, in most cases, not to simply read and study it but to use it. Open source code derives its value not from its static qualities but from its *living* ones.^{[194](#)}

If a developer is looking for help with a programming problem, they might come across an answer on Stack Overflow where someone's written out exactly the code they need. That code is in static state: a commodity available for anyone to consume.

But when a developer copy-pastes that code into their own software, it suddenly comes to life. It might break instantly. It might break their other code, and now they have to rewrite everything around it. When software transitions from static to active state, it starts to incur a set of hidden costs.

Code that runs in production, like C, is in active state. It's a living organism, like a tree in a forest. It depends on other things, and other things depend on it to survive.

When Richard Stallman first described free software as “free as in speech, not free as in beer,” the distinction he wished to make is that the term “free” referred to what one could do with the software, rather than to its price.[195](#)

At a conference many years later, Jacob Thornton, the developer who cocreated Bootstrap, suggested that open source is, instead, “free as in puppy”:

Open-sourcing something is kind of like adopting a cute puppy. You write this project with your friends, it’s really great, and you’re like, “OK, like I’ll open-source it, it’ll be fun! Like, whatever, we’ll get on the front page of Hacker News.” . . . And it is! It’s super fun, it’s a great thing.

But what happens is, puppies grow and get old, and pretty soon . . . your puppy’s kinda like a mature dog . . . and you’re like, “Oh my god, so much time is required for me to take care of this thing!”

. . . If someone had told me a month before I open-sourced Bootstrap that I would have 40,000 stars and that I would quit fucking Twitter and I would still be spending hours a night looking at issues, I would’ve been like, “LOL, yeah right, no way, this little thing?”

. . . I’ve created endlessly more and more projects that have now turned into dogs. Almost every project I release will get like 2,000, 3,000 watchers, which is enough to have this guilt, which is essentially like, “I need to maintain this, I need to take care of this dog.”[196](#)

Whether free as in speech or beer, Stallman was describing code in terms of static state. Thornton, on the other hand, identified how code behaves

differently in active state: it's more like a puppy that requires care and feeding over time. Puppies are never “done,” so long as they are alive. Buying a puppy is not like buying furniture; bringing a living creature into one’s home signifies the beginning of a new set of responsibilities.

It’s unsurprising that free and early open source developers like to talk about the freedom to fork. Forking is a useful exit strategy when we ignore the dependencies and upkeep required by the project in question. In reality, a widely used project makes itself unforkable because it’s become more than just the code.

After one unhappy user of core-js, a popular Javascript library, commented on the project, “Mind you, at any time any open source package can be forked and ads can be removed this way. I’m not threatening, just saying,”¹⁹⁷ another user responded with amusement: “Forking is one thing, maintaining is another ;).”¹⁹⁸ Denis Pushkarev, the project’s maintainer, chimed in, “Feel free to fork and maintain it - I will only be glad that core-js maintenance now it’s [sic] not my problem.”¹⁹⁹ Maintenance costs make the difference between “forking as a credible threat” and “forking as a desirable outcome.”

Code, while it’s being traded, appraised, or exchanged, assumes its static form, with all the properties that we’d expect of a commodity. But once it finds users, code springs to life, switching to an active state and incurring hidden costs. To quote Norbert Wiener, the mathematician who pioneered the field of cybernetics, “Information and entropy are not conserved, and are equally unsuited to being commodities.”²⁰⁰

THE HIDDEN COSTS OF SOFTWARE

We can think about software as having three major types of costs: *creation*, *distribution*, and *maintenance*.²⁰¹

Creation is frequently powered by intrinsic motivation. These are the fixed “first-copy” costs. Distribution is mostly powered by the platforms used by creators. These costs are low or free due to supply-side economies of scale. But maintenance is still a mystery. Maintenance costs typically fall onto the creator but, as we’ve seen, are not intrinsically motivated.

Software incurs ongoing maintenance costs, both *marginal* (costs that are a function of its users) and *temporal* (entropy, or costs associated with decay over time).

Information goods are thought to have zero or negligible marginal cost, meaning that, while the first unit is expensive to produce, each additional unit costs little to the producer. (Information goods are commodities whose value is derived from the information they contain, such as articles, books, music, or code.) If I publish code to GitHub, it should make no difference to me, from a cost perspective, whether ten or 10,000 people use it.

David Heinemeier Hansson, arguing why we shouldn’t think about software in market terms, uses zero marginal cost to demonstrate his point:

The magic of software is that there is virtually no marginal cost! That’s the economic reality that Gates used to build Microsoft’s empire. And what enabled Stallman to “give away” his free software (albeit with strings attached). The freeloaders are free! There is no practical scarcity to worry about.²⁰²

While software does have lower marginal costs compared to material goods, like cars or houses, its actual costs depend on whether we're viewing it in active or static state. Code, in static state, can be bought and sold at nearly zero marginal cost. When maintenance is involved, however, software's marginal and temporal costs begin to add up.

MARGINAL COSTS

We believe that software is zero marginal cost due to the following properties, which together imply that additional copies are cheap to produce:

- **NON-RIVALRY:** If I download code from GitHub, my decision doesn't diminish your ability to download that same code. (By contrast, if I bite into an apple and hand it to you, there is now less apple for you to eat.)
- **NON-EXCLUDABILITY:** If someone owns a copy of my code, it is difficult for me to prevent them from sharing it with others. (By contrast, if I build a theme park, I can prevent people from entering by putting up a turnstile and charging admission.)

Technology policy writer David Bollier paints a rosy picture of what he calls the “information commons,” or online information goods, including open source software, under the commonly held belief that these are non-rival goods. According to Bollier, not only is software nondepletable but it can only benefit from attracting more people:

If most natural commons are finite and depletable (forests can be clear-cut, groundwater can be drained), [information] commons. . . are

chiefly social and informational. They tend to involve non-rival goods that many people can use and share without depleting the resource.

Indeed, many information commons exemplify what some commentators have called “the cornucopia of the commons,” in which more value is created as more people use the resource and join the social community. The operative principle is “the more, the merrier.” The value of a telephone network, a scientific literature or an open source software program actually *increases* as more people come to participate.²⁰³

	EXCLUDABLE	NON-EXCLUDABLE
RIVALROUS	Private goods (e.g., cars, domain names)	Commons (e.g., forests, online privacy)
NON-RIVALROUS	Club goods (e.g., cable, subscriptions like Netflix or Spotify)	Public goods (e.g., air, open source code)

Rivalry and excludability properties of economic goods.

These two properties, taken together, mean that we tend to treat open source code as a public good. In the strictest sense of the word, however, software is not quite non-rivalrous.

As when more cars merge onto a highway during rush hour, the marginal cost incurred between the first and second software user may be negligible or even undetectable. But eventually, some user N will affect the ability of

user $N+1$ to access that same software. If ten people use the same software versus 10,000 people, its developers will feel the difference.

Physical infrastructure

Software requires physical infrastructure to reliably serve a large audience without any downtime, security attacks, or interruptions in service. Today, these costs have mostly been handed off to central providers, but that doesn't make them any less real. These companies work hard to make infrastructure costs feel invisible to the rest of us.

Content is rarely hosted by users anymore. Publishing on Medium or GitHub, for example, means creators never even have to think about hosting costs. The cost of uploading a photo or video to Instagram or YouTube is paid for by the platform.

Cloud services have made it easier to upgrade infrastructure without interruptions, if they charge for anything at all. Just as anyone can store their photos on iCloud or their files on Google Drive and pay for more storage as needed, upgrading software infrastructure can be as simple as a few clicks on Fastly, Cloudflare, or Netlify. The biggest providers, such as Amazon Web Services and Microsoft Azure, have made hosting costs very cheap or free.

That doesn't mean it can't get expensive, however. Donald Stufft, who maintains the Python package manager PyPI, estimates that its infrastructure costs two to three million dollars per year in hosting, donated by Fastly.^{[204](#)} As more developers have adopted PyPI, the project's costs have grown significantly. According to Stufft, in April 2013, PyPI used 11.84GB of bandwidth.^{[205](#)} By April 2019, that figure had increased to

4.5PB.²⁰⁶ On a smaller scale, open source developer Drew DeVault estimates that he spends \$380 each month on server hosting for his projects, which he pays for with user donations.²⁰⁷

Werner Vogels, chief technology officer for Amazon and an architect of Amazon Web Services, describes how the marginal cost of physical infrastructure can become significant at scale:

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront.²⁰⁸

Distributed denial-of-service attacks, or DDoS attacks, are one way in which the cost of physical infrastructure is still made visible. In a DDoS attack, a malicious actor intentionally floods a server with requests in order to overwhelm it and shut it down. A number of DNS providers (which manage domain names), CDNs (which host content, like files and media), and other critical services have experienced large-scale DDoS attacks. Two of the biggest DDoS attacks in history were against GitHub itself: first in 2015, then again in 2018.²⁰⁹

The high fixed costs of physical infrastructure also explain why we see supply-side economies of scale as an adaptive strategy. If software consumption were truly zero marginal cost, it would be just as easy for anyone else to maintain their own version of GitHub as it is for GitHub itself. But it's far more efficient for a single platform to manage the code, security, infrastructure, support, and whatever else comes with maintaining a software product. Developers use GitHub over GitLab not just for the

network effects but also for the former's security and reliability. The same goes for why someone would use a Google product, like Gmail or Google Docs, over that of a startup. It costs money and manpower to do these things well.

Finally, there are marginal costs associated with the developer tools used to maintain software. For example, developers might use error-monitoring software like Sentry, configuration management tools like Puppet or Chef, or incident-response tools like PagerDuty, all of which are priced according to usage.

User support

User support is a significant marginal cost associated with software, plaguing not just open source developers but the biggest technology companies, which are still figuring out how to manage unprecedented levels of adoption.

When user adoption is low, the cost of support feels trivially small, perhaps nonexistent. The vast majority of users will quietly download code without ever making themselves known.

However, as adoption grows, the once trivial cost of support can become significant. Perhaps only 0.1% of users require support. If a company has 1,000 users, then only one user needs support. But 0.1% of 1 million users is 1,000 users. Suddenly, it matters to a developer whether a thousand or a million people consume their software.

Facebook doesn't have a customer support line you can call, because they have more than 2 billion monthly active users.²¹⁰ This creates an

exceptionally challenging support problem compared to companies that transact in physical goods. Dewitt Clinton, a product manager at Google, explains the problem as such:

If you have a billion users, and a mere 0.1% of them have an issue that requires support on a given day (an average of one support issue per person every three years), and each issue takes 10 minutes on average for a human to personally resolve, then you'd spend 19 person-years handling support issues every day.

If each support person works an eight-hour shift each day then you'd need 20,833 support people on permanent staff just to keep up.

That, folks, is internet scale.²¹¹

In open source, user support frequently comes in the form of “how to” or “why” questions about the project. They’re different from bug reports in that they are considered noncritical, with value going primarily back to the users, rather than the developers of the project.

Nolan Lawson, a maintainer of PouchDB, describes his experience handling user support:

Outside your door stands a line of a few hundred people. They are patiently waiting for you to answer their questions, complaints, pull requests, and feature requests. . . .

When you manage to find some spare time, you open the door to the first person. They’re well-meaning enough; they tried to use your project but ran into some confusion over the API. They’ve pasted their code into a GitHub

comment, but they forgot or didn't know how to format it, so their code is a big unreadable mess. . . .

After a while, you've gone through ten or twenty people like this. There are still more than a hundred waiting in line. But by now you're feeling exhausted; each person has either had a complaint, a question, or a request for enhancement.²¹²

Eric S. Raymond once coined the aphorism “Given enough eyeballs, all bugs are shallow.” His point is that open source software presents an advantage over closed source software, because if more people can inspect the code it will increase the chance that more bugs will be discovered. The implication is that support can be handled in a fully decentralized manner that will distribute its costs among users.*²¹³

But as Fred Brooks wryly notes in his classic engineering book *The Mythical Man-Month*, first published two decades before Raymond made his claim, although “more users find more bugs,” this results in a type of support cost that grows, and is “strongly affected by the number of users.”²¹³ As more people use open source software, more questions will be asked, and more bugs will be found—but someone still needs to review, manage, and process these reports.

Devon Zuegel, a product manager at GitHub, refers to these as the “servicing costs” of software, noting the “asymmetry between the low cost of community participation and the high cost that others’ participation places on the leaders of the community.” She compares the problem to traffic congestion caused by automobiles, where each person wants to drive their own car, but in doing so increases the congestion experienced by others, and, eventually, themselves.²¹⁴

Community moderation

Software does not exist in a vacuum. As it acquires more users, some of those people will inevitably use it in undesirable ways.

From a legal perspective, software providers have been historically absolved from addressing this cost. Section 230 of the United States Communication Decency Act (CDA) protects most platforms from liabilities associated with the content uploaded by their users: “No provider or user of an interactive computer service shall be treated as the publisher or speaker of any information provided by another information content provider.”²¹⁶

In open source, all popular licenses contain an “as is” clause, which protects creators from liability in the case of harm arising from the code’s use. The MIT license, which is the most popular open source license for GitHub projects today,²¹⁷ states,

*THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.*²¹⁸

In practice, software providers understand that the optics of being associated with undesirable content will be detrimental in the long run. Although it is not a “hard” cost of production in the way that physical infrastructure might be (if the software breaks, nobody can access it), community moderation is a “soft” cost that is enforced by social norms (“Ignore these costs at your own risk”).

If a platform or project becomes an unpleasant place, people will eventually leave. A 2017 survey of GitHub users on open source projects found that one-fifth of respondents who experienced or witnessed negative behavior stopped contributing to a project as a result.^{[219](#)}

TEMPORAL COSTS

In addition to marginal costs, software also requires ongoing maintenance to continue running successfully, regardless of how many people use it. Rather than a function of use, these costs are a function of entropy: the inevitable decay of systems over time.

It’s not just code itself that requires maintenance either, but all the supporting knowledge that surrounds it. When code changes, its documentation must also change. The most upvoted answers on a Q&A site eventually become outdated and incorrect. Programming books and videos eventually require revisions.

Maintenance makes up a significant aspect of software’s hidden costs. A 2018 Stripe study of software developers suggested that developers spend 42% of their time maintaining code.^{[220](#)} Nathan Ensmenger, the informatics professor, notes that, since the early 1960s, maintenance costs account for 50% to 70% of total expenditures on software development.^{[221](#)}

Technical debt

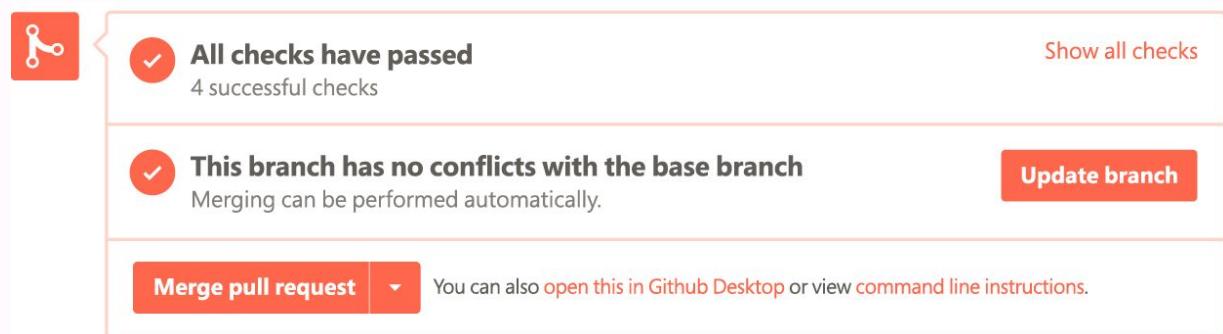
Code is “cleanest” when it’s first released, because that’s the time at which developers are thinking about the project holistically and writing it from scratch. As more code is added incrementally, software starts to become unwieldy, like a building from the 1850s that’s had new rooms, plumbing, and electric wiring added piecemeal over the years.

When adding code to a project, it can be hard to prioritize decisions on a long time horizon. A critical bug or security vulnerability might require taking shortcuts to patch things up quickly, but eventually those short-term decisions start to add up. Developers refer to this as *technical debt*: making choices that are easier today but that cost time and money to address later on.

Open source projects are particularly susceptible to technical debt because they accept contributions from developers who may not necessarily know one another, nor have full context on the project. *Scope creep* refers to the tendency for software to become bloated over time, because new code additions are evaluated incrementally rather than holistically.

Refactoring is the process by which developers pay down technical debt: rewriting, simplifying, and otherwise cleaning up the codebase without changing its functionality. Much like editing a book versus writing it for the first time, refactoring code is often dreaded and unrewarding work. Since open source developers tend toward work that they find intrinsically motivating, not only are open source projects more susceptible to technical debt but there are also fewer incentives to clean up messy code.

One of the biggest pain points for open source projects is managing test infrastructure. Software is frequently developed today using continuous integration (CI), wherein smaller code revisions are regularly merged into the master, rather than merging larger revisions less frequently. To reduce the chance of introducing breaking changes, developers use automated tests to detect whether new additions are “safe” to merge (also known as “make the merge button green” on GitHub).



A pull request that's ready to merge.[222](#)

To manage the build, test, and deployment aspects of this workflow, developers use CI services like Jenkins, Travis CI, and CircleCI. These tools are particularly important for open source projects, where developers are merging changes from people that they might not know.

But because tests are added incrementally over time—written at the time of each additional code contribution—a project’s test infrastructure can become unwieldy. It’s easy to writing a quick, imperfect test in the moment, but these small choices add up, eventually affecting everybody else’s ability to work on the codebase. There are tests that are no longer useful, tests that are slow, tests that are flaky, tests that don’t do what the developer thought it would, and tests that are orphaned by their authors.

Running all these tests before merging each pull request can take a long time. One maintainer of a large open source project told me that running his CI service took an hour and a half per pull request, yet he was expected to review and merge forty to fifty pull requests per day.

These problems affect closed source software companies, too. Alan Zeino, a software engineer at Uber, writes that, before the company upgraded its architecture, “build times skyrocketed along with our growth. With hundreds of engineers, the collective hours lost waiting for CI to build code changes numbered in the thousands every day.”²²³

Dependency management

While static code does not change over time, code in active state is closely intertwined with its *dependencies*, meaning other code that runs with it. If software is a LEGO house, each of its dependencies can be thought of as a LEGO brick. An open source project might be one of those bricks, with a different project, maintained by another group of developers, forming another brick.

Over time, code falls out of sync with its dependencies, becoming incompatible with newer versions of those dependencies. Even if your code doesn’t change, everything else around it changes. Code that hasn’t been touched in five years can’t be executed on a modern machine without some updates, unless a developer replicates its exact circumstances (also known as a developer’s *environment*), using tools such as containers or emulators. Developers might choose to stick with older versions of their dependencies, but if they ever update those dependencies something will likely break and require changes to make it all run together.

Dependency management can get particularly messy when working in a language ecosystem that prizes modularity, such as JavaScript. Npm, JavaScript's package manager, has more packages than any other popular package manager: over 1 million packages, compared to the 153,000 packages in RubyGems (Ruby's package manager) and the 175,000 packages in PyPI (Python's package manager).^{[224](#)}

In one sense, JavaScript's modularity makes the ecosystem more brittle. The surface area of potential problems is bigger, since there are more dependencies, each one managed by a different developer. On the other hand, a modular approach is theoretically more resilient: even if one package disappears, since it is smaller in scope it should be easier for developers to contain the problem and find a similar substitute.

PACKAGE MANAGER	AVERAGE NUMBER OF DEPENDENT PROJECTS (AMONG TOP 50 PACKAGES)	AVERAGE NUMBER OF DIRECT CONTRIBUTORS (AMONG TOP 50 PACKAGES)
Maven (Java)	167,000	99
pip (Python)	78,000	204
npm (JavaScript)	3,500,000	35
NuGet (.NET)	94,000	109
RubyGems (Ruby)	737,000	146

Dependencies versus direct contributors for the top packages used in popular programming language ecosystems. Note the extreme

disparity for npm.²²⁵

One particularly challenging aspect of dependency management is security. Much like technical debt and refactoring, security vulnerabilities can be time-consuming to manage, with little upside for the developer, coupled with the fear of an extremely bad situation if they miss something important. In a commercial setting, developers are paid to deal with the things they don't feel like doing. In open source, where so much work runs on personal motivation, security can easily fall by the wayside.

Security issues affect not just the code that a developer writes but those who depend on their code. The event-stream compromise, mentioned in Chapter 2, was a targeted attack not against event-stream's direct users but against the users of a digital wallet application that depended on the project.

Russ Cox, writing about "Our Software Dependency Problem," explains,

Adding a package as a dependency outsources the work of developing that code—designing, writing, testing, debugging, and maintaining—to someone else on the internet, someone you often don't know. . . .

. . . . We are trusting more code with less justification for doing so.²²⁶

Publicly disclosed security vulnerabilities are listed in the National Vulnerability Database, which is maintained by the United States National Institute of Standards and Technology (NIST), and are identifiable by a Common Vulnerabilities and Exposures (CVE) ID. But not all vulnerabilities make it into the database.

Bigger projects might use a monitoring tool like Snyk or SourceClear to scan their code and notify maintainers of known security vulnerabilities, but

the maintainers of smaller open source projects, frankly, often can't be bothered. In 2017, GitHub added the option to receive security alerts for open source projects and their dependencies, focusing on a few major ecosystems, including JavaScript and Ruby. But seeing that there is a vulnerability in one's dependency tree doesn't mean a maintainer will take the time to address it. Some vulnerabilities are easily patched; others feel like more trouble than they are worth.

Even big companies are susceptible to inertia when it comes to patching security vulnerabilities. In 2017, Equifax reported a security breach in which more than 140 million customers' personal information was compromised, including Social Security numbers, credit card numbers, and addresses. The vulnerability was found not in the code that Equifax had written but in one of its open source dependencies, Apache Struts. The security vulnerability had been disclosed with a CVE ID several months before, and a patch had been released, but Equifax's developers failed to update the company's software in time.^{[227](#)}

Adapting to user needs

Maintenance is often classified as *reactive* work: it's the minimum required to keep things running smoothly. But users' needs change over time, too. Software must also change to meet these needs, or else risk becoming irrelevant.

Clayton Christensen famously identifies and analyzes this problem in *The Innovator's Dilemma*, the 2003 book in which he tries to understand why successful companies can be overtaken by new ones, even if they are doing well. By focusing too much on iterating upon their incumbent product, companies risk missing major opportunities for so-called "disruptive innovation," which eventually replaces existing products.²²⁸

Similarly, state-of-the-art software, even with regular maintenance, will eventually be replaced by something else that better addresses modern user preferences. It's easy to write off the cost of newness as unnecessary, compared to the more urgent costs of maintenance, but good stewardship of infrastructure requires the foresight of innovation. It's not enough to maintain a tool that everybody uses begrudgingly. The first-copy costs of software often piggyback off the mature, end-of-life costs associated with an older project.

PyPy is an implementation of the Python programming language that's designed to replace CPython, because it runs faster.²²⁹ Although CPython is still the most widely used implementation of the language, over time a replacement like PyPy became necessary to consider. The rewriting of CPython as PyPy is a large undertaking: adding support for Python 3 took an estimated 10.5 person-months.²³⁰

When maintaining software, developers must decide whether to replace parts of existing code or write new tools entirely. Managing these trade-offs adds another layer of complexity to the concept of project “health” discussed at the end of the previous chapter.

The activity, popularity, and dependencies of an open source project are narrow metrics that can’t be interpreted in isolation. These metrics can tell us how a specific project is doing, but they can’t answer the question “*Should* we continue to support this project?” Knowing whether a project is worth maintaining requires a familiarity with the ecosystem acquired through experience.

In many cases, particularly in open source, where maintaining is more expensive than creating, it can be better to abandon old tools and write new ones. Sometimes, the best thing for the ecosystem is to let certain projects die—but there isn’t always consensus between developers on how to make these decisions.

Mikeal Rogers, who wrote a popular HTTP client called `request`, decided to stop supporting its development, not for personal reasons but in order to allow other, more modern tools to flourish:

The best thing for these new modules is for `request` to slowly fade away, eventually becoming just another memory of that legacy stack. Taking the position `request` has now and leveraging it for a bigger share of the next generation of developers would be a disservice to those developers as it would drive them away from better modules that don’t have the burden of `request`’s history.^{[231](#)}

Deciding when to support, and when to deprecate, existing software is challenging, particularly in the context of open source, where oftentimes there is no central governing body responsible for making these decisions. There is no formal obligation between developers to coordinate with one another, only a social expectation to work together in the best interest of the community (assuming they are even part of the same community). Nor is it easy to get users to switch to a newer project, even when the need for an upgrade has been determined.

Python’s maintainers decided to end support for Python 2 in order to nudge users toward Python 3, citing a “small but constant friction” in maintaining compatibility between the two language versions:

We are keen to use Python 3 to its full potential, and we currently accept the cost of writing cross-compatible code to allow a smooth transition, but we don’t intend to maintain this compatibility indefinitely. Although the transition has not been as quick as we hoped, we do see it taking place, with more and more people using, teaching and recommending Python 3.[232](#)

Encouraging users to move from Python 2 to Python 3—which introduces breaking changes—has been challenging. Python’s maintainers announced their decision far in advance and spent hours on outreach, partnering with companies and publishing their timeline in order to build awareness among users.

Sometimes, multiple standards simply exist alongside one another, without any clear consensus. When Setuptools was forked into Distribute, there were three similar libraries that coexisted for years—Setuptools, Distribute,

and Distutils—leading one confused Python developer to ask on Stack Overflow,

I'm not sure what's the difference between these modules—distutils, distribute, setuptools. The documentation is sketchy as [*sic*] best, as they all seem to be a fork of one another, intended to be compatible in most circumstances (but actually, not all) Could someone explain the differences? What am I supposed to use? What is the most modern solution?²³³

In response, one developer suggested, “I think most Python developers now use Distribute,” while a Distutils maintainer explained that, while Distutils “is the standard tool used for packaging,” “in some subcommunities, [Setuptools is] a *de facto* standard.” Another developer recommended Setuptools, despite noting that Distutils “is still the standard tool for packaging in Python.”

MEASURING THE VALUE OF CODE

Software’s low marginal cost has democratized how people share ideas. If distribution is cheap, code can be consumed and shared more rapidly.

Guido van Rossum, the author of Python, recalls writing ABC, a predecessor to Python, and how difficult it was to physically distribute copies to other developers: “I remember around ’85, going on a vacation trip to the US, my first ever visit to the US, with a magnetic tape in my luggage No wonder we didn’t get very far with the distribution of ABC, despite all its wonderful properties.”²³⁴ By contrast, when van Rossum released Python in 1991, he published it on the alt.sources

newsgroup, a mailing list for source code, which made distribution vastly cheaper and easier.

Zero marginal cost means that developers have free access to endless amounts of code. Any developer can find millions of repositories on GitHub for free. In addition to the educational benefits, having access to others' code helps bring developers' ideas into reality faster by reducing their fixed costs.

Instead of having to build every component from scratch, developers can access all the tools that other developers have already built and published online. “In the ’90s, if you wanted to code you had to code basic data structures first,” Nat Friedman, CEO of GitHub, explains.²³⁵ “But today developers can get much more done while writing less code.”

It’s cheaper to reuse existing software components than to write code from scratch, which also makes it possible for entrepreneurs to start software companies with fewer up-front costs. The entire software industry owes its financial success to leveraging this arbitrage.

These benefits are passed down to software’s users, too. If software doesn’t cost much to make, developers can offer consumers more of their tools, toys, and applications at affordable prices.

But software’s “zero marginal cost” property heavily favors its consumers. If software is free to consume, in terms of value accrued to the producer it is worth nothing. A developer might gain reputational benefits from creating a popular project, but those benefits are usually short-lived compared to the long-term costs of maintenance.

Over time, ignoring software's hidden, ongoing costs creates a tension between producer and consumer. As economist J. Bradford De Long and law professor A. Michael Froomkin write,

If goods are non-rival—if two can consume as cheaply as one—then charging a per-unit price to users artificially restricts distribution: to truly maximize social welfare you need a system that supplies everyone whose willingness to pay for the good is greater than the *marginal cost* of producing another copy. And if the marginal cost of reproduction of a digital good is near-zero, that means almost everyone should have it for almost free.

However, charging price equal to marginal cost almost surely leaves the producer bankrupt, with little incentive to maintain the product except the hope of maintenance fees, and no incentive whatsoever to make another one except that warm fuzzy feeling one gets from impoverishing oneself for the general good.

.... Without non-financial incentives, all but the most masochistic producer will get out [of] the business of production.^{[236](#)}

Software producers have never really figured out how to sell code itself. Stratechery's Ben Thompson makes a similar observation about the music industry: "The music industry was primarily selling plastic discs in jewel cases; the music encoded on those discs was a means of differentiating those pieces of plastic from other ones, but music itself was not being sold."^{[237](#)}

By tweaking the properties of *non-rivalry* and *non-excludability*, which make information zero marginal cost, producers artificially nudged

consumers into paying for code, tamping it into physical formats like a genie stuffed into a lamp, and selling them off together.

Code, when tethered to corporeal form—distributed, for example, on disks or CDs—is easier for producers to commoditize. Charging for books, CDs, or floppy disks containing code made software “excludable.” And, similarly to how video-rental stores like Blockbuster made movies “rivalrous” by only having a limited number of physical copies to rent, companies like Adobe made software rivalrous by selling commercial licenses with a limited number of user seats. If you wanted more seats, you had to pay for them.

We still use artificial rivalry to monetize content today. For example, although libraries now offer e-books, only a certain number of people at a time can check out the same e-book, due not to the limitations of technology but to restrictive commercial licenses.

Producers commoditized code for as long as they could. Commercial licenses, layered on top of software, leveraged the threat of legal action to get customers to pay. Digital rights management (DRM) was an attempt to control access by embedding restrictions directly into the technology. Music purchased on iTunes had limits on the number of times it could be shared with others, a constraint embedded directly into the song file.

The bundling strategy still works in some instances. Apple still commoditizes software by keeping it tightly coupled with its hardware. Big game companies, like Microsoft Xbox, PlayStation, and Nintendo, also combine game software with a hardware platform lock-in.

But it was always possible to pirate software, or to photocopy a book containing code. And as our lives moved increasingly online, code and physical form began to slide apart even further. Code by itself is not, and has never been, worth anything, and consumers already know this intuitively when they refuse to directly pay for it.

These lessons were memorably encapsulated by Bill Gates's attempts to sell BASIC in the 1970s. BASIC was the software used to run Altair, a personal computer made by a company called Micro Instrumentation and Telemetry Systems (MITS). It was Microsoft's first product, licensed to MITS and sold together with the Altair.

At one of MITS's demos, a paper tape containing BASIC was stolen. Pirated copies of BASIC began to appear and permeate through the software community, cutting into Microsoft's royalties. Gates, furious, penned his famous 1976 "Open Letter to Hobbyists," in which he notes that "less than 10% of all Altair owners have bought BASIC," and that "the amount of royalties we have received from sales to hobbyists makes the time spent on Altair BASIC worth less than \$2 an hour."²³⁸ BASIC software, once unbundled from the Altair computer, wasn't worth very much at all.

Producers constantly fight an uphill battle to threaten, lock down, beg, blame, and shame consumers into paying for content. They must play the role of cowboys, roping consumers with their lassos and dragging them in the desired direction, whether by threatening the users with lawsuits or wheedling readers into disabling ad-blocking software—which extends to not allowing them to read articles in a browser's private mode.

Although we continue to pay lip service to the idea that consumers *should* pay for software somehow, code struggles against its bonds, spurred perhaps by the famous observation, attributed to Stewart Brand, that “information wants to be free.”

Economist David Friedman tells a joke that goes like this: Two economists walked past a Porsche showroom. One of them pointed at a shiny car in the window and said, “I want that.” “Obviously not,” the other replied.[239](#)

The joke is about revealed preference: the idea that we can only understand consumer preferences based on their actual behavior. If we were to rewrite the joke about open source software, it might go something like this: Two developers cloned a popular open source project. One of them pointed at a donation button on the README and said, “That’s a great idea.” “Obviously not,” the other replied.

Code, in static form, helps us understand its value through the lens of the *consumer*: how and why code is distributed so widely. Viewed through the eyes of the *producer*, however, if we only treat code as a commodity then producers have clearly lost the war on value. Code’s value slid slowly at first, then dramatically, as more and more content flew about the internet in the ethereal form of bits.

Everybody remembers Brand’s statement that information wants to be free, but Brand also points out that “information wants to be expensive.”

Developer Ben Lesh once tweeted, “Open Source is such a strange thing. The open source work I do is clearly the most impactful work I do, but no one wants to pay me to work on it. Yet I’m asked to speak about it, and the work I’m actually *paid* to do no one really wants to hear about.”  [240](#)

Even as software's purchase value is being driven dramatically down, its social value seems to be going dramatically up. We can't live without software anymore, but we also don't want to pay for it. How is this the case?

The author Jane Jacobs explores these conflicting views in her 1961 book *The Death and Life of Great American Cities*, in which she tries to explain why urban planning policy failed cities. Jacobs's major critique of urban planning in the 1950s is that the planners treated cities—the layout of their buildings, parks, and roads—as static objects, which were only developed at the outset, rather than continuously revised according to how people used them.

To make her case, Jacobs cites Dr. Warren Weaver's 1958 *Annual Report of the Rockefeller Foundation*, which explores three “stages of development in the history of scientific thought”: problems of simplicity, disorganized complexity, and organized complexity.

Problems of *simplicity* are a “two-variable problem”: for example, calculating the pressure of a gas, which depends on its volume. Problems of *disorganized complexity* are those that deal with “two billion variables”: for example, predicting the motion of a cue ball as it moves across a billiard table.

However, Jacobs proposes that cities are actually a problem of *organized complexity*, or those “situations in which a half-dozen or even several dozen quantities are all varying simultaneously and in *subtly interconnected ways*”:

Consider again, as an illustration, the problem of a city neighborhood park. Any single factor about the park is slippery as an eel; it can potentially mean any number of things, depending on how it is acted upon by other factors and how it reacts to them. How much the park is used depends, in part, upon the park’s own design. But even this partial influence of the park’s design upon the park’s use depends, in turn, on who is around to use the park, and when, and this in turn depends on uses of the city outside the park itself.²⁴¹

To identify problems of organized complexity, Timothy Patitsas, a theologian who builds upon Jacobs’s work, paraphrases a heuristic from urban ministry practitioner Douglas Hall: “The question is, are you dealing with a toaster or a cat? . . . Can you take it apart and put it back together? It’s a toaster. If you take it apart, do you belong in jail? That’s a cat.”²⁴²

In the case of cities, Jacobs argues that the “perfect” city cannot be designed once and for all, because how people use their environment changes over time. Although we can objectively appraise the value of a home (What’s the square footage? How many bedrooms does it have?), its value is also affected by its surrounding area (How safe is the neighborhood? Are there good schools nearby?). If a house is built in an upscale neighborhood but that neighborhood later falls into disarray, the house’s value will depreciate as well.

Similarly, treating code as a living organism does not replace the idea of software as a commodity. Rather, it’s that software can be understood as *both* artifact *and* organism. The rules of the “information economy,” like patents and licenses, lend themselves well to commoditized content, but

when content is a living organism its value is better measured in terms of people and relationships.

This innate duality—software visible as both a fixed point and a line—is at the heart of today’s conflict around how we value not just software but online content more broadly. Is software worth nothing, or is it indispensable to society? The answer is both.

DEPENDENCIES

Code, in active state, carries its value in its *dependencies*, or who else is currently using it. If I publish code and nobody uses it, it’s worth less than other code I’ve written that’s embedded in software used by millions. I may derive personal value from the other code I wrote, but its value to others is negligible.

In this way, software is comparable to public infrastructure, and similar valuation methodologies apply. Like code, infrastructure derives its value from its active dependencies, irrespective of the cost of its construction or maintenance.

Imagine we need to determine the value of two bridges. One costs \$10 million to maintain, while another costs \$100 million. We don’t assume the second bridge is worth more simply because it costs more to maintain. Instead, we would evaluate which bridge provides greater public value in order to determine which is worth maintaining. As economist Randall W. Eberts, who studies public systems, notes,

The cost of maintaining the existing infrastructure is not necessarily a measure of the value of transportation capital to the economy. Even if

each previous investment decision that built the current capital stock was optimal when the project was constructed, the economy and population continue to change, which also changes the value of the infrastructure.^{[243](#)}

Infrastructure is recursively defined by public consensus. It's the set of structures that we've collectively decided are most valuable *in any given moment*, and, therefore, its boundaries and definitions are expected to change over time.

Dependencies don't give us the full story, however. While some open source code might be widely used and depended upon, it might also be trivial to maintain or easy to replace, which affects its overall value.

In 2016, a developer named Azer Koçulu, who felt unhappy with npm over a naming dispute, decided to take down all his modules without warning, declaring in a blog post titled “I've Just Liberated My Modules” that “NPM is someone's private land where corporate is more powerful than the people, and I do open source because, Power To The People.”^{[244](#)}

One of those liberated modules was a library called left-pad, a simple function that would right-justify your text. It was just seventeen lines long. But because several other large npm packages used left-pad—including Babel, a JavaScript compiler downloaded 11 million times per week—suddenly, thousands of developers everywhere started hitting errors in what was described by the media as a “break the internet” moment.^{[245](#)}

The effect was certainly disruptive. In terms of dependencies, the fact that so many developers immediately felt its absence suggests that left-pad was widely depended on. But was it valuable?

Npm published a postmortem of the affair, in which they noted that a developer named Cameron Westland “stepped in and published a functionally identical version of left-pad” within ten minutes. After some developers continued to hit versioning errors (Westland’s version was listed as 1.0.0, but some dependencies had specified an older version of left-pad), npm itself “took the unprecedented step” of restoring Koçulu’s original library, which he had deleted. While npm’s users continued to debate the ethical and policy implications of this decision for months afterward, the time between Koçulu “breaking the internet” and the restoration of normal operations was just two and a half hours.²⁴⁶

Left-pad was widely depended upon, and its disappearance had noticeable effects, but it was also trivially easy to replace. This property is known as *substitutability*—the ability of goods or services to be replaced by other alternatives—and it applies to a lot of open source code, particularly as one goes higher up the application stack.

While left-pad is an extreme example, software substitutability helps us understand why frontend web frameworks like Angular or Vue are harder to monetize than are databases like MongoDB and MySQL. There are lots of frontend frameworks to choose from (though switching costs substantially increase after you’ve chosen one to build with), but I’m less likely to want to make my own production-quality database.

Substitutability applies to other online content, too. Consider how many “10 Ways to Maximize Your Productivity”-type blog posts are published to LinkedIn every week, or how many “chill music” playlists you can swipe through on Spotify or YouTube, with little preference for the specific songs

or artists involved. Each of these blog posts or playlists might attract thousands or millions of users, but they are also easily substituted.

Open source code also tends to be a highly *elastic* good, meaning that its consumers are sensitive to price changes and restrictions, and will happily switch to a competitor if needed. (Airline tickets, for example, are an elastic good, while health insurance is inelastic.)

Developers are finicky consumers. Not only are they discerning, with a high degree of sensitivity to slight differences between open source projects, but if they don't like the options presented to them, they're frequently inspired—and have the ability—to try making their own version. By contrast, if I need to buy a table for my home but don't like any of my options, it's unlikely that I'd try to build my own table from scratch.

Because creation is intrinsically motivated, developers are always iterating upon existing software, tweaking similar versions to their own liking. Not only do *copies* of open source projects replicate like a virus, then, but developers enjoy coming up with more originals, each one slightly different from the next. It's infuriating for anyone who hopes to monetize open source code, because software's rapid creation cycle reduces code's scarcity and increases its substitutability. Someone might have written and published a software library that's pretty good, but if I have access to a hundred other free libraries that are a lot like it, it's unlikely I'll want to pay for it.

In addition to dependencies, then, we can use a counterfactual approach to assess the value of widely depended-upon code:

- If this code didn't exist, how much would it cost to accomplish the same thing in a different way?^t
- How much money and time does this code save us?
- What does this code allow us to do that we couldn't have done otherwise?

The counterfactual approach accounts for not just the tangible costs but the opportunity costs of software, also known as *non-marginal effects*, or “the potential for infrastructure to support and enable economic activity and output that may not otherwise have been possible.”²⁴⁷

REPUTATION

From a public interest perspective, if we wanted to make a list of all the open source code that's most important to support, the aforementioned approach is a reasonable start. If open source code is like infrastructure, we want to measure its value based on a combination of *dependencies* (Who uses the code?) and *substitutability* (If this code disappeared, how hard would it be to replace?). It's just as hard to measure the economic value of public infrastructure as it is the value of open source code, so these methodologies are more art than science.

But measuring the value of code based on dependencies only gives us one part of the equation. It matters who *uses* open source code, but doesn't it also matter who *makes* it?

In Chapter 3, we looked at how contributors to an open source project are not fungible. A casual contributor doesn't carry the same value as a maintainer, because it takes time for a new developer to get up to speed. The *tacit knowledge* required to maintain a project—meaning knowledge that is difficult to externalize and transfer to others—suggests that the perceived value of open source code is also at least partly a function of who's behind it.

Sindre Sorhus is a JavaScript developer who maintains, by his estimate, over 1,100 npm packages,²⁴⁸ which are, in the aggregate, downloaded 2 billion times per month.²⁴⁹ He's also the creator of a number of other popular projects, including Awesome (curated lists of topics ranging from games to databases, and one of the most-starred repositories on GitHub)

and Refined GitHub (a popular browser extension that adds custom improvements to GitHub's interface).

As of 2018, Sorhus was working full-time on open source, thanks to sponsorships managed through Patreon and GitHub Sponsors (a platform-native sponsorship product launched by GitHub in 2019). It's true that his sponsorships aren't commensurate with the value he adds to the open source ecosystem overall. (Graphtreon, an independent Patreon analytics site, suggests he's made less than \$4,000 per month on Patreon since June 2018.²⁵⁰) However, Sorhus keeps his expenses low, living abroad rather than in his native Norway. In a 2015 blog post, he explains, "I don't really care much about money or material things, don't use much money, and don't have a lot of monthly expenses."²⁵¹ In 2017, he told an interviewer he was living in Thailand and "would be fine with" less than \$1,500 per month.²⁵²

While there are a lot of things about Sorhus's life that we could remark upon, what's relevant to this discussion is that he raises money on Patreon not to work on a specific open source project but to work on open source more generally.

All told, it's obvious that Sorhus has written a lot of open source code that's widely depended upon. We could easily measure his value based on how many people rely upon his code.

But it also seems that developers sponsor Sorhus not just based on dependencies but on the basis of Sorhus himself. After all, there's plenty of open source code that's just as widely depended upon but that doesn't pull in the same number of sponsorships as Sorhus. It's Sorhus's visibility, and his reputation, that make it possible for him to raise money through

sponsorships. Like dependencies, the *reputation* of a developer is a dynamic property, based on their past and expected future value, viewed through the eyes of others.

The practice of paying maintainers to work on open source projects is not new. Donald Stufft, for example, who maintains Python’s packaging tools, was hired by Hewlett Packard Enterprise,²⁵³ and then Amazon Web Services, to improve and maintain Python’s packaging.²⁵⁴ A company could just contribute their own employees to a given open source project, and they often do, but some also hire and pay maintainers. The maintainer’s reputation translates into value for the company, whether it’s brand association or having a direct line to someone with influence on the project.

What does seem newer, however, is the practice of open source developers raising money from their fans and users, independent not just of a salaried job but of a specific project. Drew DeVault, for example, runs a Patreon campaign to “work full time on free and open source software,” listing multiple projects that he’s responsible for.²⁵⁵ And Evan You’s Patreon page supports his personal work on the web framework Vue.js (donations go “directly to support my full-time effort on Vue”),²⁵⁶ which is separate from money raised for Vue.js on Open Collective (those funds are used “to support major undertakings by core contributors, and sponsor community events”).²⁵⁷ While Evan’s ability to raise money on Patreon is certainly tied to the reputation he has built as the author of Vue.js, his Patreon funding is distinctly personal.

When assessing the value of an open source project, we typically focus on dependencies, but, increasingly, we need to assess the value of *who* produces that code. Thus we find Sophie Alpert, a prominent React

developer, bewilderedly reporting that she was offered \$600 to open a pull request on a random open source project due to the visibility she would bring, prompting her to jokingly wonder aloud, “Is this what it feels like to be an influencer?”²⁵⁸

Consider the experience of coming across an article that you read and found useful. If you were to describe the value of that article, you could measure its page views, or the number of people who are talking about it online (*dependencies*). You could talk about whether anything else like it exists (*substitutability*). But if you really liked the article and felt compelled to support it somehow, you might also take a closer look at who wrote it. A one-time op-ed written by the CEO of a multibillion-dollar company might be enjoyable, but not something you feel the need to pay for, because it’s unlikely they’ll continue to regularly write similar posts in the future. By contrast, a thoughtful research piece from an independent writer might compel you to subscribe to their newsletter or sponsor their work, in hopes that they’ll write more of the same kind of material.

A reader’s relationship to an author is not fungible. The same article carries different value depending on who wrote it, as well as the reader’s interest in, and expectation of, reading other work by that person in the future.

Velocityraps is the pseudonym of an Egyptian streamer named Mostafa, who, over the course of one NBA season, livestreamed more than 1,000 NBA basketball games to hundreds of thousands of viewers. Instead of using advertisements, he posted donation links, accepting money via Bitcoin and Venmo. Mostafa made between \$15,000 and \$20,000 in donations in less than a year, which he claimed was more than he could make in his town of Port Said as a mechanical engineer.²⁵⁹

What's particularly interesting about this example, however, is that Mostafa was illegally streaming NBA games that sports fans could theoretically have watched simply by paying for a cable subscription. Their willingness to pay for the same basketball game varied depending on who provided the content. As one sports fan, Ryan Regier, observes in a Medium piece, "This is a fascinating revelation about illegal streamers. Even though people can acquire content for free from them, *they still give them money.*"²⁶⁰

We measure the reputational value of content differently depending on whether we're viewing it in static or active state. "Likes" and "follows" are not the same reward on social media. A viral tweet might gain thousands of likes, but those don't necessarily translate into follows. The same goes for views and likes on a YouTube video, versus subscribing to its creator's channel.

The difference between likes and follows also helps us understand when maintenance costs matter. A viral YouTube video doesn't necessarily have ongoing costs if its creator doesn't plan on making more videos. But if the creator aspires to become famous for their work, they also become, in a sense, a maintainer, because they need to keep producing more content in order to maintain their reputation.

We can think of a creator's reputation as a "battery," or store of value, for attention. More followers mean more attention in the bank, but when people follow a creator they do so because they expect to receive more content. If creators don't produce anything new, their followers will eventually get bored and leave. Reputation, like software, requires maintenance over time.

* In practice, there is scarce evidence that this is the case, and quite a bit of evidence that it likely is not, because most people just use

software instead of carefully inspecting the code. The benefit of “extra eyeballs” has a maximum beyond which additional reviews are not useful.^{[215](#)}

^tThis is not dissimilar to *shadow pricing*, where, in the absence of a market price, value is calculated based on what consumers would be willing to pay to obtain a particular good.

MANAGING THE COSTS OF PRODUCTION

05

“*Nobody can keep open house in a great city.*”

—JANE JACOBS, *The Death and Life of Great American Cities*²⁶¹

When explaining why nobody wants to pay for software, people often cite the *free-rider problem*, which is the idea that if you can't exclude others from consuming a good they'll use it without paying. Eventually, the good becomes overused, as producers lack the resources—usually provided by consumers—to supply it.

It's easiest to see how free-rider problems apply to non-excludable, rivalrous goods, a situation better known as the *tragedy of the commons*. If a public park is free to access, people will use it without paying for maintenance and upkeep. As more people flock to the park, its quality is diminished. The trash cans will overflow, the crowds get packed, the grass ground down to mud. To address this problem, we typically pay for public parks with our taxes; some national parks also charge an entrance fee.

But when it comes to *public goods*—i.e., goods that are both non-excludable *and* non-rivalrous, like software—it's a bit harder to see where the free-rider problem applies. After all, a thousand people can read the same article, or use the same snippet of code, without diminishing its quality.

In the physical world, public goods are usually provided by the government, paid for by taxes: street lights, national defense, clean air. Should the government provide our open source software? Most developers will probably read that sentence and scream a resounding “NO!”^{*}

Economists tell us that we tend to rely on governments to provide our public goods because otherwise they'll become *underproduced* over time, meaning that consumers are unlikely to provide them on their own, due to the free-rider problem.

But in the online world, we don't have governments to provide our public goods. Open source software, in particular, often involves developers from different countries committing code to the same project. If an open source project has one developer in Australia and another in India, which government's job is it to support production? Whose laws govern the project's code?

Early lessons from open source cryptography help to illustrate why governments are ill-suited to meet the regulatory needs of open source developers.

In the 1970s and 1980s, many software developers used the Data Encryption Standard (DES) to encrypt digital data. DES was adopted by the United States government as an official standard in 1977. But the National Security Agency (NSA) also changed the DES to make it weaker so they could break it, if needed, with a sufficiently powerful computer. As other countries started making their own encryption standards, developers decided they wanted something more secure than the DES.

In 1991, a programmer named Phil Zimmermann released an encryption program called Pretty Good Privacy (PGP). But cryptography, because it overlapped with national security, was considered a form of munitions in the United States. If cryptographic code crossed the borders to another country, it was treated as a munitions export.

Early open source cryptographers, like those writing OpenSSL, had to become licensed arms dealers to be able to write and “export” (i.e., distribute) their code. As a result, Zimmermann found himself under criminal investigation by the United States government for distributing PGP, although no charges were ever formally filed.

At the time, the US State Department ruled that, while code on a floppy disk could not be exported, books *containing* code—protected under freedom of speech—were permitted. So Phil decided to publish the same PGP code as a book, titled *PGP Source Code and Internals*, which he could then send around the world.

Eventually, the United States dropped most export controls on open source cryptography. But the extreme disparity between regulation and reality highlights how, while governments provide public goods and services in our physical world, it’s much more difficult for them to serve this same role in our online world. Open source code is transnational, affecting developers and users across many different countries, whereas governments are beholden to national interests (such as national security). And the world of software moves so fast that the law can’t keep up.

In the absence of government, we don’t yet have tidy answers to how online public goods are built, maintained, and paid for. These questions extend to the regulation of technology more broadly, as our physical governments continue to butt heads in a digital space, whether it’s over the European Union’s GDPR (General Data Protection Regulation) laws or Google’s now-terminated Project Dragonfly, a search engine designed for China. When it comes to software and other forms of online content, it’s not clear

whose governments are in charge, if anyone is. We have to find new ways of thinking about the problem.

The need for a new approach is why Elinor Ostrom's writing has gained renewed appeal in recent years. Her work provides a framework for understanding how people can self-manage non-excludable resources without resorting to outside institutions like the government. Before we throw our weight behind solutions, however, we first need to revisit the problem.

The classic economic example of an underproduced public good is a fireworks show. If I shoot fireworks from my house, all my neighbors can enjoy them, because I can't stop them from viewing the fireworks. However, I might eventually get a little tired of buying materials, organizing the show, and absorbing the legal risks.

Early open source advocates and scholars like Eric S. Raymond and Yochai Benkler point out that the problem is not quite that simple. In the physical world, the cost of providing material goods often makes it prohibitively difficult to provide public goods "for the fun of it." The United States spends nearly \$700 billion per year on national defense, for example.²⁶²

But online, it's not that expensive to write code, record a video, make music, or publish one's thoughts. Intrinsic motivation fuels our ability to provide this content to others. We make it not because we're calculating the dollars and cents involved, but simply because we *want* to.

Our desire to make things is visible throughout our lives, not just while working at a company or collaborating with others, but as a part of who we are as humans. We like asking questions. We like to think out loud. We like

to tinker, whether with words, code, or imagery. Sometimes, we’re lucky enough to find other people who want to make things with us, but the urge to create is innate to each of us individually.

What’s more, people want to *share* the things they make. When we talk about the desire to “express ourselves,” it’s not just about externalizing something trapped inside us, but also about having that part of us understood by others.

We can’t contain the ways in which people want to make things: it happens everywhere, all the time. We also can’t prevent people from sharing things: information wants to be free, and it’s beneficial to both creators and their audiences that it remain widely available.

Whenever I come across the fireworks show example, I find myself wondering whether the assumed outcome is reflective of actual human behavior, or of the joyless nature of economists. I think about the houses in the suburban Pennsylvania neighborhood I grew up in, whose owners put up a glittering swath of Christmas lights every year: sledding Santas, galloping reindeer, dancing candy canes. Decorating your house for the holidays is a suburban American tradition. Every year, driving home around Christmastime, we’d approach my neighborhood and my mom would beg my dad to take the long way home so we could drive past all the houses dressed up for the holidays and marvel at their shiny, blinking toys.

My neighbors put up their decorations knowing that others would see them. There’s a reason why the decorations are on the outside of the house, not just the inside. But they did it for themselves, too. They did it because it’s fun to design the perfect holiday show and provide Christmas cheer to others (and maybe one-up the Joneses next door).

Economists aren't entirely wrong about where this goes sideways, however. Imagine if my parents and I, instead of driving past our favorite Christmas house and murmuring in awe, pulled into our neighbor's driveway and banged on the door, demanding that he bring back the surfing Santa. We were hoping for last year's tropical Christmas theme, but this year he went with a traditional look. Where are the palm trees?!

Our neighbor decorates his house for Christmas because, for whatever reason, he loves to do it. It costs him nothing if others want to enjoy those decorations. But if his neighbors started knocking on his door and putting in requests, or telling him how to decorate his house, he'd probably get annoyed and eventually stop putting the decorations up.

When it comes to online public goods, I think our struggle to come up with provisioning solutions is due to the fact we haven't defined the problem clearly enough. Put another way: When software is in static state, *what if there is no free-rider problem?* When code is non-rivalrous, it only has first-copy costs, which the creator is intrinsically motivated to provide—so the problem doesn't seem to lie in how many people *consume* it.

For this reason, I'm skeptical of attempts to charge for access to information, whether it's newspaper articles or open source software. If some people like to make things of their own accord, and most content is a highly substitutable good, it's unlikely that online content will ever be underproduced. Someone out there will always want to make something for the rest of us. Why prevent anyone from consuming that content, given the enormous social benefits of making it freely accessible?

Instead, I'd flip the question to ask: *What if, rather than being underproduced, software is actually overproduced?* The problem with the

Christmas lights isn't that anyone can drive by and view them. A problem only surfaces if we think our neighbor owes us anything; if we cross the invisible boundary and knock on his door, making demands and requesting changes.

Similarly, it doesn't make sense to charge for access to code or content, because it's not that too many people are consuming the good itself. It's the production—not consumption—of software that suffers from too much demand. Our expectations about unfettered online participation have caused consumers to spill over to the other side.

PRODUCTION AS A ONE-WAY MIRROR

Only an extremely rude person would consider marching up to our neighbor's house, knocking on the door, and demanding a surfing Santa. But online, it's considered normal for people to comment on articles and posts with derogatory or demanding things, or to open issues on an open source project, requesting new features or asking for support.

In Chapter 4, when examining the costs and value of software, I suggested that code can be viewed in two ways: both in static and active state, and as a function of both dependencies (who's using it) and the maintainer's reputation (who's making it).

I'd like to extend these ideas one step further and argue that *managing open source code requires separating its production from consumption*: treating them as not one but two types of economic goods. Anyone can consume code, but only a limited number of people can produce it.

Guido van Rossum, the author of Python, stepped down in 2018 after twenty-seven years as the language's BDFL.²⁶³ His decision was widely attributed to the fatigue he experienced from a proposal called PEP 572, which suggested syntax changes to Python, and which ballooned into long, branching discussion threads that attracted commenters who didn't have sufficient context to participate in the conversation.²⁶⁴

The bigger issue, he explains in a blog post, was that Python's participatory decision-making process didn't scale:

Discussing PEPs on python-dev and python-ideas is clearly not scalable any more. (Even python-committers probably doesn't scale too well. :-)

I wonder if it would make sense to require that for each PEP a new GitHub *repo* be created whose contents would just be a draft PEP and whose issue tracker and PR manager would be used to debate the PEP and propose specific changes.

This way the discussion is still public: when the PEP-specific repo is created the author(s) can notify python-ideas, and when they are closer to submitting they can notify python-dev, but the discussion doesn't attract uninformed outsiders as much as python-{dev,ideas} discussions do, and it's much easier for outsiders who want to learn more about the proposal to find all relevant discussion.²⁶⁵

Van Rossum's suggestion highlights the critical difference between content that is *public* and content that is *participatory*. Content can be made available for anyone to read and consume, but that doesn't mean it needs to be open for anyone to participate. Much of the fatigue that open source

developers experience comes not from making their code public but from expectations around making their code participatory.

Open source code, in static state, is a public good, meaning that it is both non-excludable and non-rivalrous. Like my neighbor's Christmas decorations, if it can be consumed at nearly zero marginal cost we should just let people have it. The value of this code can be measured like any other type of infrastructure: by its number of dependencies, as well as by its substitutability.

The *production* of open source code, however, functions more like a commons—meaning that it is non-excludable and rivalrous—where *attention* is the rivalrous resource. Maintainers can't stop users from bidding for their attention, but their attention can be depleted.

My neighbor is willing to allocate a certain number of hours to planning and putting up his Christmas decorations. When I knock on his door and make requests, or otherwise try to “help” him, I draw from the attention he's allocated to decorating his house, which is just one of many things he has to do with his day. When attention is over-appropriated—say, there's a crowd of people lined up outside, ringing his doorbell—a tragedy of the commons occurs, in which my neighbor no longer wants to put up his annual Christmas decorations (a tragedy indeed).

The value of the commons is derived from the reputation of its members, meaning the person or people who produce it. If my neighbor were to stop putting up his decorations, I would be a poor replacement. I have no idea which lights to buy or how best to arrange them. Thus, my neighbor is worth more to the commons than I am. If we put it to a vote, our neighbors

are much more likely to want him putting up Christmas decorations than me.

Open source code itself is not a common pool resource but a *positive externality* of its underlying contributor community. Users can consume, or “appropriate,” code at zero marginal cost, because what the commons actually manages is not code but attention. When developers make contributions, they appropriate this attention from the commons.

Ostrom’s principles for managing the commons only apply when there are clear group boundaries, which help reduce the risk of over-appropriation by setting and enforcing meaningful social norms. But if anybody can appropriate a maintainer’s attention by making demands, the commons will eventually be depleted.

To reduce the over-appropriation of attention in open source software, we can think of its production as a *one-way mirror*, where we design for the parasocial, or one-sided, relationships that are endemic to stadiums, rather than for the interpersonal relationships that are associated with clubs.

Users may consume code all they like; their public conversations might indirectly influence future iterations; and they can even “see into” the production side (for example, by having read-only access to mailing list discussions, chat logs, and pull requests). However, it’s mutually understood that they do these things from a respectful distance.

Jonathan Zdziarski, an iOS developer also known as NerveGas, hints at what a one-way mirror pattern could look like:

There is definitely a place for users and their demands, however that's not inside the community (unless they're also contributing devs); the community, as in practicing any art form, is vulnerable; you wouldn't sit and criticize a painter while they're still painting their piece. The user base needs to be moved outside of the artistic realm and into the museum, where your software is on display.^{[266](#)}

Zdziarski's comments echo those of Kristen Roupenian, the author of the viral short story "Cat Person," cited in the introduction of this book, who remarks, "I want people to read [my book]. I hope they like it. And, at the same time, I don't want to know what they think about it."^{[267](#)}

When production is a one-way mirror, creators are shielded from distraction, building things in public view but without the expectation that they engage with unhelpful contributors. We can think of this as a form of read-only access, not unlike the permissions system of open source projects themselves.

As if watching a fireside chat, outsiders can observe the interactions *between* people in a community without participating in them. In open source, we might imagine that anybody is welcome to read community mailing lists, issues, pull requests, and chat logs, but further participation requires the approval of existing developers (or, in some cases, paying for "write access").

The one-way mirror pattern is used by a number of online communities today, such as Lobsters, a computing-focused community,^{[268](#)} and Product Hunt, a community for discovering new products.^{[269](#)} These communities are public for anyone to read, but in order to post new users need an invite from an existing user.

Jonathan Zdziarski proposes moving to a model that he calls “peer source,” where all his projects would become private repositories, which only trusted developers (people he knew, or that someone could vouch for) could access. “The rest of the community can download binaries, and have the satisfaction that there is accountability on some level, just not to them,” he writes.^{[270](#)}

The idea of “peer source” is similar to private, invite-only torrent communities, which arose as a way to avoid the over-appropriation of file sharing. Peer-to-peer file sharing works like a commons, in which files are the common pool resource. There are seeds (people who upload content, a.k.a. “producers”) and leechers (people who download content, a.k.a. “consumers”).

When sharing is public, it’s possible to end up with too many leechers and not enough seeds, given that users have no incentive to upload their own content—only to consume it. In a private community, however, users are required to maintain a certain upload/download ratio; otherwise they’ll get kicked out.

The difference between file sharing and open source, however, is that code, like other forms of online content, *can* be consumed irrespective of how it’s produced. Peer-to-peer file sharing requires seeds. It doesn’t function as a one-way mirror, because consumers always impose a marginal cost onto producers. But users can consume open source code without affecting producers, so long as they don’t appropriate their attention.

Typically, a common pool resource is both produced *and* consumed solely by its members: a closed system of users who are also contributors. But online content can also be consumed at no cost by outsiders. The ability of

non-members to consume code should be treated as a positive externality, rather than a sign that they are part of the community. Tourists are allowed to visit Paris, but they're not Parisians.

Returning to the example of my neighbor's Christmas decorations, it's not that he's putting them up *for* us. He does it because he enjoys it (and because it makes him look good). Our ability to view his decorations is a positive externality of whatever decisions he's chosen to make. We don't get to tell him what to do, but we do get to enjoy the benefits of his work.

In open source, anybody should be able to not only view, download, and fork open source code, but also to witness the interactions between members. There's no reason to prevent users from accessing any of this information, so long as they don't appropriate attention from producers.

When attention is being appropriated, producers need to weigh the costs and benefits of the transaction. To assess whether the appropriation of attention is net-positive, it's useful to distinguish between *extractive* and *non-extractive* contributions.

Extractive contributions are those where the marginal cost of reviewing and merging that contribution is greater than the marginal benefit to the project's producers. In the case of a code contribution, it might be a pull request that's too complex or unwieldy to review, given the potential upside. Or it might be an enthusiastic user who wants to organize a community event that requires more resources from the project's developers than it's worth. The most common forms of extractive contributions are comments, questions, and feature requests.

C. Titus Brown, a maintainer who's involved in the Python community, describes extractive contributions as such:

Extractive contributors are far more of a burden than [sic] you might think. They consume the effort of the project with no gain to the project. Sometimes feature requests, questions, and high-energy discussions lead the project in new, worthwhile directions, but quite often they're simply a waste of time and energy for everyone involved.²⁷¹

Brown notes that even experienced, active contributors can make extractive requests: for example, a company might request certain features from an open source project, but then doesn't provide resources for its ongoing infrastructure and maintenance. Citing a friend, Brown says that a simple way to explain extractive contributions is that "sometimes pull requests are more effort than they are worth."

Withoutboats, a contributor to the Rust project, describes how innocuous contributions can become extractive as the project grows:

When there have already been 770 comments on a subject, you're obviously not going to read them all. . . . Every comment added to a discussion thread is ultimately a form of debt, and its [sic] a form of debt with compound interest.

. . . . No matter how many GitHub issues we create, it seems that every single one will grow in length until it becomes an unsustainable conversation. We have a problem of induced demand: just as adding lanes to a highway does not resolve traffic congestion, creating more threads does not resolve comment congestion.²⁷²

The threat of induced demand explains why the solution to managing extractive contributions is not always to add more contributors—tempting as that may be—but to draw clearer boundaries between which types of tasks are worth allocating *anyone*'s attention to and which can be safely ignored. Rather than asking “How do we do *everything* that’s required of us?,” the better question is “How do we assess what’s *most* important to do?”

In the case of Rust, maintainers could recruit a new contributor to read the comment threads and summarize them for the group, but the real answer might be to stop reading comments altogether. One Rust maintainer, describing the futility of managing his notifications, compares it to “holding down a mudslide.” Instead of responding faster, it’s often wiser for maintainers to reduce demand in the first place.

By contrast, *non-extractive* contributions are those that produce a net benefit for the project’s developers. Users who don’t interact with the project’s developers also don’t impose a marginal cost. As Brown explains,

Users are interesting, because they contribute nothing to the project but also cost us nothing. If someone downloads sourmash [a command-line tool and Python library], installs it, runs it, and gets a result, but for whatever reason never publishes their use and cites us, then they are a zero-cost user.²⁷³

Casual contributions are non-extractive when they resemble Benkler’s *modular, granular* tasks, which require little additional input from maintainers and are inexpensive to review and merge. Bigger, more substantial contributions can also be non-extractive, if the benefit of the contribution exceeds the cost of reviewing and merging it.

Translations are one example of how a contribution can be either extractive or non-extractive, depending on how it is managed. A common contribution for newcomers to a popular project is offering to translate documentation into another language. At first glance, it seems additive: Why not make the documentation available in more languages? It's also relatively self-directed (the maintainer doesn't speak the language, so they have to trust the contributor to handle it themselves), and relies on specialized skills from the contributor (they're presumably fluent in the language they're translating into), all of which make it seem like the ideal contribution.

But documentation is not static. As the primary language's documentation changes over time, translations will also need to stay synchronized. Will the contributor who's offering to translate documentation be willing to continue maintaining it indefinitely? Probably not. Does the maintainer want to track down new contributors who can keep their documentation up-to-date in Dutch, Slovenian, and Portuguese? Probably not.

It's possible for a maintainer to sink hours into coordinating and synchronizing multiple translations, with little benefit back to the project. So unless a maintainer *wants* to manage that work, it's more likely that they'll either accept a one-time translation from an eager contributor and clarify that it won't be maintained or they just won't accept translations at all.

Extractive contributions are still considered contributions, and they're hard for maintainers to ignore, because, as Brown puts it, "In the open source world, developers are taught to value all users, and will often bend over backwards to meet user's [sic] needs." Nobody wants to be accused of appearing unsympathetic or unwelcoming. This fear of invoking public

wrath leads maintainers to feel pressured to review contributions, even when they are extractive.

Best practices for managing the commons, however, suggest that maintainers *should* avoid allocating attention toward extractive contributions. In Ostrom's language, this is the equivalent of members ignoring outside input (whether from the government, the market, or other opinionated outsiders) on how to manage their shared resource, if the advisor is not part of the community that governs it. Not everybody can participate, regardless of enthusiasm or good intentions, because if producer attention is not carefully managed it will be depleted.

The programming language Clojure is an example of an open source project that explicitly draws boundaries around participation. Rich Hickey, its author, is known for his top-down approach to governance, which other Clojurists sometimes disagree with but generally seem to respect. In response to a blog post titled "On Whose Authority?," written by a developer criticizing Hickey's governance style, Hickey wrote a lengthy comment on Reddit challenging the idea that Clojure's development needs to be a community effort at all:

Authority comes along with authorship . . . I don't know why that is no longer obvious. Thinking otherwise yields a broken economic model, where people are not entitled to control over the products of their own labor, and thus are without control over their livelihood.

Clojure was not originally primarily a community effort, and it isn't primarily one now. That has to be ok. The presumption that everything is or ought to be a community endeavor is severely broken.^{[274†](#)}

In the sections that follow, I'll combine Chapter 3's discussion of social dynamics with Chapter 4's explanation of the costs of production, in order to explore how open source maintainers leverage these principles to both allocate their attention wisely and realize value from their work.

MANAGING PRODUCER ATTENTION

There are a few typical patterns used by open source maintainers to manage their available attention:

- Reduce up-front costs
- Make themselves less available
- Distribute costs onto users
- Increase total attention available

While we might think of stadiums in terms of scarcity, and clubs and federations in terms of abundance, I've observed all of the above strategies used across project types, regardless of how many contributors they have. A single maintainer might direct their users toward a user-to-user support system, while a large project might use automation to reduce overall demand. As explored in Chapter 3, "work done" is a function of total attention available, not a project's number of contributors.

In addition to costs fluctuating over the life of a project, the benefits of maintaining also fluctuate, which in turn affects how much attention a maintainer is willing to allocate. Maintainers might gain more benefits in the earlier stages of a project and pay more attention to it as a result. But if a maintainer has already reaped the reputational benefits of being associated with the project, in the absence of additional rewards the amount of attention they're willing to allocate will decrease over time, especially if the work required is growing.

For any given project, we can imagine a point where the marginal benefit of maintaining the project crosses the marginal cost of doing so. For example, when support volume becomes too high for a given project, it's not worth it

for the maintainer to continue responding to each additional issue. They will either prioritize which support issues they respond to, bring on new maintainers to help, or push support entirely onto users. Each of these strategies is a way of reducing a maintainer's costs below the threshold of the benefits they receive.

Maintenance costs are anathema to software developers. Every developer tries to reduce the amount of maintenance work they need to do over time. But software is never done, and also never dies: the cost of maintaining a project can be near-zero, but it is asymptotic.

We can't ever completely abstract away the cost of maintenance, because software still degrades over time, regardless of how many people use it. Developers can only reduce those costs enough that it feels sustainable, find new ways to dedicate time to maintenance, or hand off the project to a new maintainer.

At a high level, it's also possible that throwing away projects and starting fresh is the most cost-effective approach: rebuilding the whole system, piece by piece. Maybe it's not the maintenance of any one project that we're trying to solve for, but the broader ecosystem. Maintainers can, and often do, mark projects as "unmaintained" and ask users to fork the code if they want to make changes. Just as important as maintaining software is making it easy, and socially acceptable, for existing developers to step down and move on.

Finally, perceived costs and benefits are subjective. One maintainer might enjoy spending many hours on certain tasks that another maintainer would not. Some maintainers care more about security vulnerabilities or code readability than others do. Others might refuse to spend time on support,

even if they got paid for it. We might have opinions on what we wish maintainers would do, but what they ultimately choose to do depends on their own desires and motivations.

REDUCING UP-FRONT COSTS

Automation is a developer's best friend when it comes to reducing work that doesn't require the attention of humans at all. It's the equivalent of setting up filters for your inbox so that unnecessary emails won't vie for your attention in the first place.

It's not just open source developers but software companies more generally that make heavy use of automation, particularly in the realm of product support and content moderation. In a 2019 report, Twitter estimates that 38% of offensive tweets are now preemptively flagged through automation, instead of through the company relying solely on manual reporting.^{[275](#)}

Jack Dorsey, Twitter's founder and CEO, explains that automation is at least partly necessary to meet rising demand: "We want to be flexible on this, because we want to make sure that we're, No. 1, building algorithms instead of just hiring massive amounts of people, because we need to make sure that this is scalable."^{[276](#)}

In open source, developers use *automation* to reduce costs, whether it's bots, tests, scheduled tasks, linters and style guides, canned responses, or issue templates, all of which help manage the contribution process. Even documentation is arguably a form of automation, anticipating answers to common questions and making them publicly available.

E. Dunham, a former Rust maintainer, describes automation as a way of working with, rather than against, humans themselves: “The process of moderation cannot and should not be left to a computer, but we can use technology to make our mods’ work as easy as possible. We leave the human tasks to humans, but let our technologies do the rest.”²⁷⁷

Another way that maintainers reduce costs is through what Kraut and Resnick would call *screening mechanisms*, including the choice of developer tools. While some maintainers move to GitHub in order to boost their projects’ appeal and attract new contributors, others seem to deliberately enjoy not being on GitHub, because it makes their projects harder to get to.

Kraut and Resnick emphasize that screening mechanisms aren’t just about “weeding out potential members who are not a good fit for the open source project”; they’re also about “increasing the commitment of those who attain those privileges.”²⁷⁸ Screening mechanisms are a “double-edged sword,” which “may drive away some potentially valuable members who are unwilling to make the initial investment.” Some maintainers cite similar concerns around adding issue templates, checklists, and error messages: they don’t want to deter new contributors by making the process too complicated.

On the other hand, when developers feel that they’ve earned their contribution, they’re more likely to stick around. Whether maintainers should gatekeep, versus opening the floodgates, depends on their current needs. They can choose to dial contributions up or down based on the level of demand.

Finally, maintainers can reduce their up-front costs by making certain actions more expensive. They could charge money to open an issue or pull request, similar to how highways charge a higher toll during commute hours to reduce the number of cars on the road. We haven't yet seen many developers experiment with these ideas, partly because it's still seen as violating the participatory norms of open source, and partly because it would require platform support to pull off well. (GitHub doesn't currently make it possible to charge for these actions.)

As discussed in Chapter 1, platforms play a critical role in absorbing costs for creators, in categories such as distribution, hosting, or security. By reducing or removing these costs, platforms enable creators to focus on their ideas.

Sometimes, creators run into conflicts with the platforms they depend on, because it's not always clear whose job it is to handle which tasks. Content moderation, for example, is a hotly contested battle on social platforms like Facebook and Twitter. Platform policies can affect creators' ability to produce what they want, whether it's Instagram's policies on nudity, YouTube's policies on content deemed unacceptable to advertisers, or Twitter's policies on hate speech.

Platforms don't cover the costs of labor, either: you still have to bring your own pickaxe to the gold mine. In 2016, nearly 2,000 maintainers signed a letter called "Dear GitHub" (uploaded, of course, to a public GitHub repository), criticizing the platform for not helping with the growing costs of maintenance:

Those of us who run some of the most popular projects on GitHub feel completely ignored by you. We've gone through the only support

channel that you have given us either to receive an empty response or even no response at all. We have no visibility into what has happened with our requests, or whether GitHub is working on them.^{[279](#)}

In response, GitHub staff made a pull request to the repository, where they apologized—“We hear you and we’re sorry”—and acknowledged, “Issues haven’t gotten much attention from GitHub these past few years and that was a mistake, but we’ve never stopped thinking about or caring about you and your communities.”^{[280](#)} Over time, both the dear-github repository and the isaacs/github repository—started by npm’s founder and former CEO, Isaac Schlueter, as “just a place to track issues and feature requests that I have for github”—became de facto places for open source maintainers to track their feature requests for GitHub, although they received little formal attention from GitHub for years.^{[281](#)}

Because solutions are often limited by platforms, the ability of open source maintainers to reduce their costs depends in large part on what GitHub officially supports. However, there are still a number of examples to look at.

Continuous integration and automated tests

Continuous integration is the practice of deploying smaller, but more frequent, changes to production, which helps developers iterate more rapidly on software.

Deploying changes more frequently requires more coordination among developers, because multiple people are changing the same codebase. A number of automated services have sprung up to manage this pipeline, including Travis CI and CircleCI. Even GitHub has started offering its own

continuous integration services through GitHub Actions, a system for building custom automated software workflows.

Continuous integration is used in software development more widely, but, as discussed in Chapter 4, it's particularly useful in open source because developers must coordinate with other contributors that they may not know personally. By setting up automated tests and workflows for building, testing, and deploying code, maintainers can feel more confident about merging contributions from unfamiliar developers, so long as they pass the checks. And contributors can feel more confident about submitting changes to a project they might not be deeply familiar with.

Bots

Bots are automated, configurable assistants that have become indispensable to large open source projects. They can perform a variety of services, whether it's being the first to respond to an issue, checking that contributions pass tests and comply with requirements, or updating dependencies. Rust's Highfive bot, for example, finds the best reviewer for a given pull request then tags them,²⁸² while Kubernetes's Prow Robot reviews, triages, and merges pull requests.²⁸³

Researcher Mairieli Wessel et al. found that out of 351 popular, active projects sampled on GitHub, 26% use bots to manage their workflow.²⁸⁴ While some maintainers worry about bots coming off as impersonal to new contributors, this form of automation seems to be an increasingly common practice: Wessel et al. identify a “boom” in bot adoption starting after 2013.

The screenshot shows a GitHub issue thread with the following sequence of events:

- stale bot commented on Nov 20, 2017**:
This issue has been automatically marked as stale because it has not had recent activity. It will be closed if no further activity occurs. Thank you for your contributions.
+ 😊 ...
- stale bot added the wontfix label on Nov 20, 2017**:
Bad bot, bad bot! Shut up and sit down, doggy dog :D
1 ⚡ 1 😊
- stale bot removed the wontfix label on Nov 20, 2017**:
This issue has been automatically marked as stale because it has not had recent activity. It will be closed if no further activity occurs. Thank you for your contributions.
11 😊

One bot comes to another bot's defense on GitHub.[285](#)

Code style guides and linters

Developers don't just test their code for functionality but also for the way in which it is written. As when multiple authors contribute to the same book, every developer has their own style of self-expression, even when creating a shared project with others.

Open source projects, especially those with many contributors, will enforce a "style guide" to keep code clean and make it easier for newcomers to read and understand. $1+2$ and $(3^2)/(471-468)$ might both produce the same answer, but one is easier to read than the other.

Linters are used to analyze code for errors and can help enforce code style automatically, whether it's a preference for semicolons or an affinity for tabs over spaces. Code style tends to follow the preferences of the project's author.

Templates and checklists

Templates for issues and pull requests are like contact forms for customer support: they help "prescreen" user requests and filter out noise. Among the top hundred projects on GitHub by issue volume in 2018, 93% of projects used issue templates.²⁸⁶ The README for the command-line program `Youtube-dl` explains why templates are helpful:

Before we had the issue template, despite our extensive bug reporting instructions, about 80% of the issue reports we got were useless

`youtube-dl` is an open-source project manned by too few volunteers, so we'd rather spend time fixing bugs where we are certain none of those

simple problems apply, and where we can be reasonably confident to be able to reproduce the issue without asking the reporter repeatedly.²⁸⁷

Maintainers sometimes use checklists to make sure that contributors have read and agreed to all their conditions before submitting a pull request. React Native Firebase, a collection of React Native modules that connect to Firebase services, asks those filling out a bug report to add a fire emoji to the subject line, which proves they read the entire template:

<!— Thanks for reading this far down  —>

<!— High quality, detailed issues are much easier to triage for maintainers
—>

<!— For bonus points, if you put a  (:fire:) emojii [sic] at the start of the issue title we'll know —>

<!— that you took the time to fill this out correctly, or, at least read this far
—>²⁸⁸

LIMITING AVAILABLE ATTENTION: THE N=1 APPROACH

In addition to using automation to reduce costs, maintainers also manage demand by reducing their available attention. Philip Guo, who maintains an open source project called Python Tutor, describes this approach as “n=1”:

Whenever something goes from n=1 developer to n=2, that is the biggest jump, because then it becomes a collaborative software development project with multiple people. I can't keep everything in

my head anymore, I need to coordinate and communicate with others . . .

So, in general, I don't accept pull requests. I may look at them if you make pull requests, and I may comment on them, and I might get inspired by some of your ideas, but I do not have a workflow set up so that I, you know, accept contributions from outside . . . not because I don't like to, but just because I do not have the time to be able to vet and maintain all that.²⁸⁹

While Guo acknowledges that he's likely missing out on opportunities to collaborate, by keeping the work localized to just himself he's been able to happily maintain the project for many years.²⁹⁰ He also doesn't worry as much about code quality or documentation, because he considers it a personal project, on which he is the only developer and which he doesn't expect to pass on to anyone else.

Developer Gabriel Vieira recalls a similar experience when asking Roberto Ierusalimschy, who authored the Lua programming language, if he could submit a pull request to the project:

I once asked [Roberto] if I could send a PR about a feature I wanted in the language and he said something that I never forgot: “Yes, but I won’t use your code. I love that people send me ideas, but I actually enjoy coding... so I will gladly take your suggestions, though I will write it myself.”

He then explained this “dictatorial” behavior is what allowed him to keep the implementation simple and concise over the years. At the time he boasted about the source code being less than 8K LOC [lines of code], though it has likely increased in recent versions.²⁹¹

In both cases, Guo and Vieira acknowledge that pull requests can function more like comments or suggestions. These contributions can still provide helpful inspiration to the maintainer, without the technical and social coordination work that comes with merging others' code into their project.

Homebrew's lead maintainer, Mike McQuaid, limits his available attention by using a tiered approach to managing contributions:

- **FIRST-TIME CONTRIBUTORS** need documentation, templates, and automated checks “to encourage high quality pull requests,” as well as positive reinforcement when they do get a pull request merged, which encourages them to contribute again.²⁹²
- **SECOND-TIME CONTRIBUTORS** warrant a mild increase in a maintainer’s attention, such as providing more detailed code review. Mike encourages maintainers to “make more ambitious requests” of the contributor, rather than trying to fix the contributor’s problems themselves.
- **THIRD-TIME CONTRIBUTORS** “now warrant . . . individual, focused attention,” including providing mentorship, suggesting additional areas of contribution, and vouching for their work to other community members.

For a first-time contributor, the burden is on them, not the maintainer, to submit a “clean” pull request that meets the criteria for review. Maintainers don’t need to invest deeply into casual contributors until they come back for at least a second time. Repeat contributors increasingly gain the trust and attention of maintainers.

However, maintainers need to “help contributors help themselves” by setting up systems that make it easy for a contributor to know when they

have succeeded. Maintainers should give contributors the tools they need to self-serve, such as clear documentation, tests, checklists, and bots to guide them through the process.

DISTRIBUTING COSTS: USER-TO-USER SYSTEMS

User-to-user systems are a way of distributing costs without appropriating attention from producers. Depending on the task, sometimes users are even *more* motivated to participate and manage the work than maintainers themselves are.

Moderation

Moderation can become expensive and unwieldy as user adoption grows, a pain experienced by many popular social platforms. When managed entirely by employees, content moderation does not scale to Instagram's 1 billion monthly users,²⁹³ or Facebook's 2 billion monthly users.²⁹⁴ But a user who's strongly affected by inappropriate content is intrinsically motivated to take action. Enabling these users to flag and hide content can help reduce moderation costs.

Platforms must provide tools not just for reporting problems but also for enabling users to control their experience. GitHub has dedicated internal teams who manage spam and abuse reports. However, GitHub also encourages projects to manage their own community behavior. The community guidelines state, “We rely on our community members to communicate expectations, moderate their projects, and report abusive behavior or content. We do not actively seek out content to moderate.”²⁹⁵

GitHub provides a suite of tools that allows maintainers to moderate their communities, whether it's by locking the comments on controversial discussion threads, blocking users, or banning them from the project. Twitter takes a similar approach, encouraging block and mute features to customize each user's own experience. While the quality of these tools inspires endless debate among their users, the underlying strategy is an effective way of distributing and reducing moderation costs.

The screenshot shows a GitHub issue thread. At the top, a comment by user **michaelficarra** is shown, indicating they closed the issue on May 22, 2018. Below this, another comment by user **ljharb** is shown, indicating they deleted the **smoosh** branch on May 22, 2018. The main body of the thread has a large grayed-out area with a lock icon in the center. Above this area, there are two buttons: "Write" and "Preview". Below the grayed-out area, a message states: "This conversation has been locked as **too heated** and limited to collaborators."

Example of a locked issue thread on GitHub.[296](#)

Enthusiastic community members can also serve as a “shield” to creators, removing unnecessary distractions and allowing them to focus on their work. Creators with large followings start to recognize familiar faces. Instead of trying to moderate everything themselves, they can designate trusted members of their communities to help.

Platforms should recognize the value of these relationships and find ways to officially support them. Twitch, for example, recognizes an official “moderator” role, which complements the role of the “broadcaster,” or streamer. According to Twitch’s guidelines, moderators “ensure that chat is

up to the standards of the Broadcaster by removing offensive posts and spam that detracts from conversations.”²⁹⁷

Twitch encourages broadcasters to choose moderators from their communities, such as “viewers that are recommended by people you trust, or by several other Mods. . . . If you are able to notice users that are helpful to other users, and show up to your streams often, you should talk to them about Modding and see if that is something they would want to help with.”

These broadcaster-moderator relationships parallel the behavior of open source developers. John Resig, creator of the JavaScript library jQuery, confesses,

The very first person I brought on to the jQuery project wasn’t another developer to help contribute It was someone to help manage our community. Because we were just getting so much feedback, so many issues coming in, and so many people using us, we needed a way to kind of keep track of it all, and ensure that people’s concerns were being heard. And so being able to kind of delegate some of that responsibility meant that I could spend more time focusing on writing code.²⁹⁸

User support

Even with all the resources in the world, companies like Google, Apple, and Facebook can’t meet the cost of user support alone. Instead, they rely on public support forums.

Poorly implemented support forums can feel like ghost towns, where users shout questions into the void in hopes that someone is paying attention. But

when enough people ask and answer questions, users are often in a better position to help one another than an official support team is.

In open source, maintainers frequently push user support questions onto forums like Stack Overflow or group chats like Discord or Slack, where users can help answer one another's questions. In 2018, I analyzed a set of the top one hundred open source projects by issue volume, and found that 89% were using something besides GitHub issues to manage their support needs, listing an average of two additional channels. The most popular channels were dedicated forums (41%) and Stack Overflow (38%), followed by IRC (22%), Gitter (20%), and mailing lists (18%).²⁹⁹

These support channels are often driven by users who, like community moderators, derive satisfaction, and sometimes reputational benefits, from helping others. They tend to operate like satellites, away from the GitHub repository where core developers congregate, but they still reduce the amount of support that maintainers have to do themselves.

When implementing both user-to-user and automated systems, there will always be *some* coordination work involved to set up and manage these systems. Moderation needs can be partly resolved by algorithms and self-motivated users, but they will always require some degree of human involvement, if only to review appeals and arbitration. However, the cost of implementing and managing these systems is often far less than the cost to maintainers of doing it all themselves.

MEETING DEMAND: INCREASE AVAILABLE ATTENTION

Even after reducing costs, limiting available attention, and distributing costs among users, there is probably still some work left for maintainers to do. In

the absence of money, ideally it's work that maintainers are motivated to do —or, at least, the additional effort required is less than, or equal to, the benefits they receive.

Once again, content moderation is an example of how not all costs can be reduced through automation, nor off-loaded onto users. Even after employing the tactics discussed above, companies like Facebook still hire teams of paid moderators to sort through disputed content. In Sarah T. Roberts's *Behind the Screen: Content Moderation in the Shadow of Social Media*, she quotes Max Breen, a pseudonymous content moderator for an unnamed tech company, who notes that this kind of work is extremely non-fungible, even among human moderators: "I don't think [outsourcing] works because . . . the culture is so dramatically different when you are trying to apply a policy that is based on [Western] culture."³⁰⁰ Breen continues, "They tried outsourcing to India before I was there, and it was such a disaster that they just canceled it and said we have to do this all in-house."

If a maintainer wants to increase the available attention for a given project, they can either bring on more active contributors or find ways to increase the attention they personally can expend on the project (for example, by raising money, or asking for more dedicated time at work).

Researcher Xin Tan looked at the effects of the “multiple-committer model” in Linux’s i1915 subsystem, finding that maintainers who give commit access to other regular contributors are able to significantly reduce their workload, given the presence of “capable candidate committers,” maintainers and committers who trust each other, and “mechanisms to ensure patch quality.”³⁰¹ In one case, a maintainer who was solely

responsible for 90% of the workload was able to reduce their share to 30% by distributing commit rights to others.

Maintainers can also manage new, significant contributions by asking contributors to work on their own “branch” first. Major features for big projects tend to work this way: a developer works on a new idea as an entirely separate project, keeping maintainers informed along the way, until it’s ready to be merged into the core project. In this way, promising, but expensive, contributions can be de-risked without appropriating attention from the commons.

Django’s developers created an Official Projects Program to incubate substantial new ideas before bringing them into Django:

With . . . the increasing number of projects living outside of the core repository that are nonetheless central to the Django community and mission, and a desire to not grow the size of the main Django repository with optional or extra code, it becomes apparent that there is a need for “official” Django projects that exist outside that repository but still are part of the Django organisation and management structure.³⁰²

Whether a maintainer increases their own available attention, versus growing the number of contributors—as well as whether they’re able to pursue either option at all—depends on the project in question. As identified in Chapter 1, there are a number of factors that affect whether a project can attract more contributors. The scope, maturity, dependencies, and substitutability of the project also affect whether maintainers can attract sufficient resources to increase their own attention on the project.

These options are not mutually exclusive either. The maintainer of a project with low contributor growth, while not building a large community of contributors, might be able to find just one more dedicated contributor who makes all the difference. Sometimes, having a less popular project can make it strangely easier to find that person, as their affinity to the project is stronger.

For projects that want to aggressively grow their contributor base, maintainers will invest more in fostering a brand and sense of community, adopting measures such as speaking at conferences, encouraging developer conversations, and maintaining an active online presence. They might embrace liberal contribution policies that default to accepting new contributions, or give away commit access more freely.

Conversely, in a project with high contributor growth, individual developers might find it easier to increase their own time on a project. For example, 92.3% of contributors to the Linux kernel are paid by an employer, for which the Linux Foundation offers this explanation: “Kernel developers are in short supply, so anybody who demonstrates an ability to get code into the mainline tends not to have trouble finding job offers.”³⁰³

Maintainers who find themselves at the helm of a stadium project can pursue similar strategies, whether it’s getting dedicated employer time or contract work, monetizing the project directly, or raising money from sponsors. Smaller projects tend to have a harder time with this, because they’re not as well recognized, but it does happen.

Henry Zhu, maintainer of Babel, was given time from his employer, Behance, to work on the project, before leaving his day job to raise money from sponsors to work on open source full-time.³⁰⁴ Daniel Stenberg,

maintainer of cURL, considers his former employer, Mozilla, to be a “primary sponsor of the curl project, since that was made up of them allowing me to spend some of my work days on curl.”³⁰⁵ He also gives credit to his subsequent employer, wolfSSL, which hired him to work on cURL, including building a pipeline of commercial support services for the project.³⁰⁶

THE ROLE OF MONEY IN OPEN SOURCE

When I first started exploring this space several years ago, I started with the hypothesis that open source maintainers are under-resourced relative to the value they provide.³⁰⁷ After publishing my initial set of research, I’ve heard from thousands of people sharing the full gamut of ideas on how to pay maintainers, ranging from promising to wacky. Throughout this experience, I’ve been frustrated that the conversation feels so scattershot.

We still don’t have a common understanding about *who*’s doing the work, *why* they do it, and *what* work needs to be done. Only when we understand the underlying behavioral dynamics of open source today, and how it differs from its early origins, can we figure out where money fits in. Otherwise, we’re just flinging wet paper towels at a brick wall, hoping that something sticks.

I want to have better conversations about money and open source that are based on what developers actually do, rather than on what we hope or wish or think they do. I’m equally tired of hearing “Let’s pay all maintainers!” as I am “Paying maintainers will destroy open source!” There is no simple answer as to where money meets open source. Knowing where money does

or doesn't apply requires getting to know the project in question: which levers are available to pull, as well as what those levers even are.

There is no one-size-fits-all solution here, because open source has matured to the point that, while its distribution licenses have been standardized, we can no longer tell a unified story about how it is produced. Sometimes, the reason why a maintainer can't find funding for their work is not because their users are mean and ungrateful, or because the world is cruel, but because that person is shy about marketing themselves. Or maybe they're hard to work with. Or their users don't actually trust them. As discussed in previous chapters, the value of a project varies widely based on its dependencies and substitutability, as well as the maintainer's reputation. Some open source projects are venture backable, while others have no business value. Every maintainer's story utilizes a similar set of variables, but they're all set to different values.

That being said, I would be remiss if I concluded this book without an explicit discussion about where money fits into open source. So, drawing upon everything we've covered thus far, let's get into it.

The discussion that follows is not a practical guide to funding, nor is it an exhaustive account of all the different ways that projects get funded, nor is it an analysis of today's funding landscape. I'm not interested in cataloging what's happening right now, but rather what could or should happen, given a deeper understanding of how open source is produced and maintained.

My goal here is to zoom out of the chatter and focus on fundamentals. Who are the potential funders of open source, and why would they pay for anything? What are they paying for? And what can an open source developer conceivably charge for?

To answer these questions, we need to return to the idea of attention as the currency of production. Historically, producers charge customers to access content. But today, producers can make content free to read, while instead charging for “write access”: meaning, the ability to appropriate attention from producers.

Intrinsic motivation can generate more attention, but so can extrinsic motivators like money or reputation. However, allocating money, reputation, or intrinsic motivation to the wrong places can be counterproductive.

In this chapter, we’ve treated the *production* of open source software as a common pool resource, where maintainer attention is both *non-excludable* (anyone can bid for their attention) and *rivalrous* (appropriating their attention reduces the total amount available).

Maintainers can make their attention *excludable* by charging for access, which turns it into a private good. Paid support, patronage, and bounties (meaning, paid rewards for certain tasks or contributions) are all examples of ways that maintainers price their attention. By making attention excludable, the quality of contributions will increase, as contributors and users compete for the attention of producers.

WHO FUNDS OPEN SOURCE DEVELOPERS, AND WHY

There are two types of funders that care enough to spend money on open source: *institutions* (usually companies, but also governments and universities) and *individuals* (usually developers who are direct users).

The popular sentiment among open source developers is that companies should underwrite the costs of development because they have the most resources. While this is probably true, based on how we currently talk about open source, my view is that choosing the right source of funding depends on the value proposition. In this section, I'll try to make the case for considering how open source projects might successfully raise money from individual developers, too.

What companies like to fund

Companies are mostly interested in open source code itself as a “product” or commodity. As a result, companies tend to value benefits like code quality, influencing the project’s roadmap, and brand association.

Code quality includes things like code security, reliability, and regular releases. In other words, companies are paying for peace of mind. They want to know that the code they use won’t break or have something terrible or unexpected happen overnight.

Daniel Stenberg, the cURL maintainer, was once contacted by a company for help after they ran into problems with a firmware upgrade rollout that used cURL. In an interview, Stenberg recalled, “I had to explain that I couldn’t travel to them in another country on short notice to help them fix this . . . because I work on cURL in my spare time and I have a full-time

job.”³⁰⁸ He sent a friend to fly out instead. (These sorts of requests are not unlike the “pay per incident” model offered by companies like Microsoft, where customers pay a flat fee per incident to receive immediate technical support.³⁰⁹)

In terms of business models, the desire for code quality tends to manifest as paying for support services or service-level agreements (SLAs). Tidelift, a company that partners with open source projects to offer commercial support to enterprise customers for their dependencies, charges for exactly that.

Dual licenses and “open core” models (akin to a freemium model, where most of the code is freely licensed, except for one key part) can also be understood as methods of charging for the code itself. The developers of Oni, a text editor, decided to dual license their code on a “time delay” for a new version of the editor, Onivim 2. Its code lives under a proprietary license that requires purchase for commercial use, but every commit converts to open source, under an MIT license, after eighteen months. In this way, Onivim 2’s developers can freely license older versions of the project while charging for access to the newest, most desirable version of the code.

Paying for code quality tends to work better for inelastic goods, where few substitutes to the project exist. If a project is highly substitutable, it’s more likely that companies will just switch to whatever comparable alternative exists, assuming that switching costs aren’t prohibitive.

While a specific developer might be well equipped to maintain the code, ultimately what companies are paying for is code quality. This makes it slightly harder to make the case for funding a specific individual.

Sometimes, companies will maintain their own private forks of a project, rather than making upstream contributions, if they perceive it's more cost-effective to do so. Or, they'll hire or divert their own employees to contribute to a project, rather than hiring or paying a specific outside contributor.

Companies will also pay for *influence* and *access* to a project they care about. It can be difficult for a company to attract the attention of a project's maintainers; they find it frustrating when issues and pull requests go unanswered.

Sometimes, this lack of attention is deliberate. Not all maintainers want to pander to the requests of corporate users, especially if their work is unpaid. I've heard from a few engineers from large companies who confess that they've "laundered" their pull requests through non-work email addresses, because being associated with their employers can make it harder to get their contributions accepted.

In terms of business models, this value proposition translates into having a direct line to the project's maintainers, whether it's paying for priority attention on the company's issues and pull requests or being able to ask questions and have maintainers walk a company's engineering team through certain product decisions. These relationships are usually not widely publicized. One popular project hosts regular "office hours" with its highest-paying supporters as a perk of sponsorship. Another well-known maintainer, who asked me to remain anonymous, quietly works on retainer to provide corporate support for his open source projects; his hourly rate translates into a comfortable salary. Much like a lake that requires paid licenses from fisheries to reduce overfishing, an open source project can

require companies to pay to “appropriate” attention from the project’s developers.

Paying a maintainer for their attention can also translate into full-time hires. Companies who hire open source maintainers are paying for a guaranteed, ongoing relationship to someone who can directly influence the project.

Depending on the arrangement, paying for access can raise governance issues, since contributors can pay their way into a project instead of earning influence through the work they put in. For example, some open source projects are governed by a software foundation, where corporate stakeholders pay an annual fee to get representation on the board (also called a “pay-to-play” model). These relationships seem to especially plague bigger, federation-style projects.

Some foundations take explicit steps to combat this behavior. The Apache Foundation is structured specifically so that individuals only represent themselves, rather than their employers.^{[310](#)}

Finally, companies also pay for *brand association* with a project, either to generate positive brand awareness among customers or to attract quality engineering talent. In terms of business models, this value proposition tends to manifest in the form of sponsorships and advertisements.

Advertising is a highly contentious topic in open source, even more than in the general public sphere. Developers value their privacy, an uncluttered user experience, and the “quiet space” that coding provides. Still, the numbers suggest there is untapped potential, if only someone could find a way to do so without ruffling feathers. Eric Berry, who started an open source advertising company called CodeFund in 2017, grew to over

\$24,000 in revenue in four months. Then GitHub intervened, prohibiting CodeFund from placing ads on READMEs, as that violated the platform's terms of service.^{[311](#)}

Two years later, Feross Aboukhadijeh ran an experiment on one of his projects, StandardJS, where he displayed an ad in the console, explaining,

The most common funding models – donations, README sponsors, or paid consulting – only work if a maintainer can get their appeal in front of users. This usually goes in a README or on a website.

But reliable, error-free transitive dependencies are invisible. Therefore, the maintainers are invisible, too. And, the better these maintainers do their job, the more invisible they are. . . .

Maybe ads aren't the answer – fine. *But telling maintainers to bury their appeals where no one bothers to look is not the answer, either.*^{[312](#)}

While Aboukhadijeh received plenty of complaints, he also gained quite a bit of public support, suggesting that developers are becoming more receptive to maintainers raising funds through advertising, even if they haven't yet reached consensus on how to get there.

Brand association via sponsorships tends to appeal to smaller companies, like consulting shops whose business relies on a particular open source project, as well as to bigger companies that might have financial resources but lack the brand to attract engineers.

Trivago, a travel company, started sponsoring webpack in 2017. After two years of sponsorship, Trivago's head of frontend engineering, Patrick Gotthardt, reflected on the experience:

All the *public visibility* you have given us lead [sic] to a situation where we suddenly became one of the most interesting companies to work for as a JavaScript developer. . . .

We've hired a lot of really great engineers who mentioned during their job interview that *our sponsoring for Webpack was one of their primary motivations for applying.*³¹³

As in all sponsorships, it can be difficult to define the ROI, or return on investment, of these opportunities. Compared to paying for code security or reliability, there's usually an upper limit to how much a company wants to pay for brand sponsorships, and these companies are more likely to churn over time.

What individual developers like to fund

Companies are generally interested in the project's code over its developers. Viewed through this lens, individual developers are clearly not as well positioned to support open source projects. An individual developer might spend five or ten dollars per month to support their favorite project, whereas a company can easily spend thousands. (Aboukhadijeh recalls a moment, walking through an open source conference, where he asked a company representative how much they paid for their booth: "They weren't sure if it was \$10,000 or \$20,000."³¹⁴) To that end, open source developers sometimes offer two monthly tiers for recurring funders, which are clearly aimed at different audiences: one that costs around ten dollars or less, for individuals, and one that costs around \$500 or more, for companies.

It's easy to cynically rest on our laurels here and conclude that individual developers could never fund projects to the extent that companies can. But

the opposite argument has been made in politics, where politicians who fund their campaigns from grassroots donations are generally viewed more favorably by the public than are those who are funded by corporate donations. I’m not sure that open source is so different.

While most open source developers don’t yet have the luxury of making these decisions, in the long run it seems more appealing to use a project that is community-supported, rather than one that’s funded by a single major company. CHAOSS (which stands for Community Health Analytics Open Source Software), a Linux Foundation project that focuses on metrics for community health, calls this the “elephant factor”: a measure of how dependent a project is on a small set of corporate contributors.³¹⁵

Evan You, who authored Vue.js, notes that Vue is frequently compared to other, corporate-backed frontend frameworks, like Angular or React, whose users “just feel more comfortable using something backed by a big company”:

To that, I usually ask them, “What do you actually think it means for an open source project to be backed by a big company?,” and they say, “Oh, it’s more stable.” You know, because they rely on it that this project wouldn’t die all of a sudden. . . .

. . . . The thing about company-backed open source projects is that in a lot of cases . . . they want to make it sort of an open standard for a certain industry, or sometimes they simply open-source it to serve as some sort of publicity improvement to help with recruiting. . . . If this project no longer serves that purpose, then most companies will probably just cut it, or (in other terms) just give it to the community and let the community drive it.³¹⁶

A common argument against funding from individual developers is that, regardless of intent, there just isn't enough money out there to sustain the salary of a core developer. Again, this is a surprising assertion, given that software developers are generally well paid. If politicians can fund entire campaigns based on individual donations from middle-class citizens, the idea that an open source project could raise money from software developers—especially given that a single project might be depended upon by thousands or millions of people—doesn't seem to require a leap of faith.

Streamlabs, which provides tools for livestreamers, including a tipping functionality, reported \$34.7 million in tips paid out to Twitch streamers in the first quarter of 2018.³¹⁷ This is only one part of the value generated on Twitch, acquired by Amazon in 2014 for \$970 million,³¹⁸ whose streamers make money through a mix of donations, ads, sponsorships, and subscriptions. This is a platform where people pay money to watch other people play games. Ten years ago, most people would've dismissed this idea as ludicrous. Today, streaming is considered an attractive, lucrative industry.

Right now, the market for open source subscriptions and sponsorships is small, because it's new and because the argument in favor of this kind of support has primarily been based on *shoulds* instead of *gottas*. In the case of both companies and individual developers, there's the temptation to make an argument based on "fairness" or altruism: "We all benefit from this project, so we should give back." While this is a legitimate reason to give five or ten dollars per month, there's a limit to how far companies and individuals will stretch their dollar out of altruism.

Appealing to goodwill leads to what Mike McQuaid, Homebrew’s lead developer, calls the problem of “sticker money”: enough money to pay for marketing swag, such as stickers, which are eagerly consumed by developers, but not enough money to quit one’s job and work on open source full-time. I’d prefer we abandon this line of thinking around why we “ought” to fund open source, and instead focus on finding reasons that an individual developer might happily drop dollars on open source until they go broke.

However, it seems more likely that individual developers, unlike companies, would pay not directly for open source *code* but to sponsor the people *behind* the code: a function of a maintainer’s reputation rather than a project’s dependencies.

Long-term, if single maintainers don’t step down from their projects (which is what typically happens today), they need some other reward to make ongoing maintenance worth it. The typical rewards for content creators are reputational gains, which they can convert into attention (meaning, an audience). If creators want to keep making things, they find ways to convert that attention into money.

Patronage is an emerging funding system for online creators today, but it’s frequently conflated with the concept of “donations.” Patronage isn’t motivated by altruism but by an interest in following a creator’s *future* work, based on their *current* reputation. It’s more like a subscription than a donation. When an individual sponsors an open source developer, then, they’re ideally not paying for code but to feel closer to the person who writes the code.

Ben Thompson, suggesting that subscriptions could be the future of local news, defines subscriptions as such:

First, it's not a donation: it is asking a customer to pay money for a product. What, then, is the product? It is not, in fact, any one article (a point that is missed by the misguided focus on micro-transactions). Rather, a subscriber is paying for the regular delivery of well-defined value.

Each of those words is meaningful:

- *Paying*: A subscription is an ongoing commitment to the *production* of content, not a one-off payment for one piece of content that catches the eye.
- *Regular Delivery*: A subscriber does not need to depend on the random discovery of content; said content can be delivered to [sic] the subscriber directly, whether that be email, a bookmark, or an app.
- *Well-defined Value*: A subscriber needs to know what they are paying for, and it needs to be worth it.³¹⁹

Thompson notes that subscriptions only work when the content is differentiated: “After all, it’s not like it is hard to find content to read on the Internet: what people will pay for is quality content about things they care about.” But *who* provides the content is a form of differentiation in itself.

Subscriptions are uniquely suited to monetizing “the regular delivery of well-defined value.” Because subscriptions are recurring, they help solve for the shortcomings of prior funding models. Patents and royalties have long existed to incentivize the creation of intellectual property, but IP is just that: “property,” a static commodity that you own and trade.

By contrast, reputation-based funding helps incentivize the *ongoing* production of creative work, because reputation is measured by past and future expectations. As writer Tiago Forte puts it, “What people are paying for is not a bunch of text. They are paying for the perspective the writer brings to the subject.”³²⁰

Right now, there isn’t a great way for prominent open source developers to monetize their reputation other than to get hired full-time at a company. Dan Abramov, a maintainer of React, built his reputation as the author of Redux, a tool for developing JavaScript apps that is frequently used with React. Redux’s development was partly funded through Patreon. His project attracted the attention of Facebook—React’s most significant corporate maintainer—and he joined the company in 2015.

While Abramov is an example of a success story, the idea that the best possible outcome for open source developers is a corporate “acqui-hire” (a term used among startups when the employees, but not the startup itself, are acquired by another company) strikes me as a suboptimal approach, given today’s burgeoning renaissance for independent creators.

Can we imagine telling Tfue, who rose to fame by livestreaming himself playing *Fortnite Battle Royale* on Twitch, that the most he could hope for was to get hired by ESPN, or telling Camila Coelho, who built her presence by posting makeup tutorials and fashion photos on YouTube and Instagram, that she should try to find a job with *Vogue*? While these aren’t bad outcomes by any means, if “get hired somewhere” were treated as the upper bound of what’s possible, today’s world of online creators would be much less interesting. Why should open source be any different?

Many popular creators make money by transacting on platforms that have their own social status economies. Eugene Wei calls this “status as a service (StaaS)”:

That many of the largest tech companies are, in part, status as a service businesses, is not often discussed. Most people don’t like to admit to being motivated by status, and few CEO’s [sic] are going to admit that the job to be done for their company is stroking people’s egos. . . .

. . . . The solution . . . doesn’t lie in ignoring that humans are wired to pursue social capital. In fact, overlooking this fundamental aspect of human nature arguably landed us here, at the end of this first age of social network goliaths, wondering where it all went haywire. If we think of these networks as marketplaces trading only in information, and not in status, then we’re only seeing part of the machine.³²¹

Twitch, Instagram, YouTube, Twitter, and Facebook are all status-based platforms. Who provides status as a service for developers?

The academic system provides us with a blueprint of how a closed reputation economy for software might work. Researchers write and publish papers, which garner citations from other researchers, which theoretically earn the authors more negotiating power, and, eventually, tenure. But academia fails to create commensurate rewards for open source developers.

Fernando Pérez is the author of Jupyter, a set of open source tools and services for interactive computing, used by researchers and companies around the world. It is arguably one of the most impactful pieces of scientific software in circulation today. Yet Pérez, a physicist by training, had trouble finding a tenured academic position for many years.

Similarly, Tom Caswell, lead developer for Matplotlib, a data-plotting tool for Python, and Andreas Müller, core developer of scikit-learn, a machine learning library for Python, both have PhDs but are employed in non-tenure-track positions. There is a sense among developers in research-related fields that, while writing open source software can have other benefits, it doesn't fit into the academic reputation system. Writing open source software with a lot of users doesn't attract as much prestige as publishing a paper with a lot of citations.

GitHub seems like an obvious home for building and amplifying status as a developer. But while GitHub has made it easier than ever before to showcase one's identity, compared to other social platforms GitHub lags far behind.

It's still not easy to understand who a developer is, or what they do, from their GitHub profile. Profile pages highlight code activity, but they miss an opportunity to highlight educational content, questions answered, or even the projects that a developer is associated with—all of which are important ways that developers build their reputation.

“Following” an open source developer isn't a particularly meaningful action on GitHub, either, although thousands of users do it anyway. When I've asked developers why they follow other developers on GitHub, they cite respect for the craft: whether experienced or just learning a new skill, they want to see how the best do it.[322](#)

Developers already recognize one another's status. Yet it's hard to see how this behavior translates onto GitHub's platform. Following a developer's or a project's activity creates a noisy stream of stars and commits, like if Facebook showed you every friend's activity non-algorithmically.

Prominent developers including Kent C. Dodds and Suz Hinton have made a name for themselves with videos and live-coding, but you wouldn't know it from their GitHub profiles.

And it's here, perhaps, that we find the most interesting tension in the generational transition from "open source developers" to "GitHub developers." Some of today's best-known developers aren't even necessarily active open source contributors—meaning, people who commit code to open source projects—but rather educators, speakers, streamers, or just public personalities.

Cassidy Williams is a software engineer who teaches React development. She also writes a weekly newsletter, live-codes on Twitch, has a Patreon with its own private channel hosted on chat app Discord, offers classes on the online learning platforms Udemy and Skillshare, and posts viral fifteen-second videos on the social video app TikTok, as well as to her nearly 90,000 followers on Twitter. There is no other social platform more prominently associated with developers than GitHub. And yet, Williams's GitHub profile reveals hardly anything about her, except that she must be popular for *something*, given that she has a few thousand followers. Williams doesn't build her reputation on GitHub, but on every other platform around it.

Eugene Wei observes that companies offer value in terms of "utility" versus "social capital," noting that "some companies manage to create utility for a network but never succeed at building any real social capital of note (or don't even bother to try)."³²³ GitHub could be a textbook example of such an outcome.

As is the case with many other developers, Cassidy Williams's reputation is still built off the open source ecosystem. React, the technology she teaches, is an open source project. But “open source developer” doesn’t quite capture the extent of what she does.

If not “open source developers,” then what do we call these developers, and how do we reconcile their relationship to open source? One hypothesis is that “open source” is quickly becoming indistinguishable from “doing code stuff in public.” The story that binds these people together is that of creative developers becoming well known for the things they make and share with an audience, whether that’s code, videos, classes, or newsletters.

Building a public reputation often requires doing highly visible things, which is why the earliest experiments seem to skew heavily toward JavaScript, whose developers seem to attract more of a social following than do developers in other, less visible programming ecosystems. Still, there are promising examples of developers who find ways to monetize their time on open source outside of JavaScript. Jon Gjengset, for example, has a popular Rust live-coding stream that’s aimed at intermediate programmers. In response to audience demand, Gjengset started a Patreon to support his work, which he then had to suspend due to his student visa status in the United States.^{[324](#)}

Twitch made it economically viable to play games online by attaching status to it. GitHub has the potential to do the same for developers, but it hasn’t yet, possibly because we still insist on highlighting the collaborative aspects of open source over the accomplishments of any one developer, even though many people now seem to find uncontroversial the assertion that much of open source is written and maintained by single developers.

Even Eric S. Raymond, who evangelized the idea of the bazaar, concedes in a 2019 blog post that there are what he calls “Load-Bearing Internet People”—meaning “a person who maintains the software for a critical Internet service or library, and has to do it without organizational support or a budget backing him up.”³²⁵ Raymond encourages his readers to allocate a budget toward funding these developers.

In today’s world, increased social status is the expected reward for making things. It’s how platforms motivate creators to continue publishing things, beyond the initial phase. Open source developers are chronically undervalued because, unlike other creators, they’re tied to a platform that doesn’t enable them to realize the value of their work. Instead of operating quietly in the background, open source developers ought to come to the forefront again.

WHERE THE MONEY GOES

Given that open source projects aren’t really “owned” by anyone, often lack a legal entity, and are developed by multiple people, all of whom might live in different countries and may have never even met one another . . . how does funding actually come together?

In this section, we’ll look at how funding opportunities are prioritized, whether to support projects or individual contributors, and which types of contributors to fund.

Which opportunities to prioritize

Even if a funder buys into the idea of allocating money toward open source, they can quickly become overwhelmed by the opportunity: “But there are so many projects out there!” A single dependency check can pull in hundreds of open source projects. How does any funder, whether institutional or individual, know where to concentrate their efforts?

In a strange way, open source’s ubiquity is both its greatest asset and its biggest challenge. Open source is so integral to our everyday lives that it’s nearly impossible to know where to begin, or whether any one person’s actions will make a difference. In a tweet, npm’s former CTO, Laurie Voss, lamented how this problem arises while fundraising:

It is very hard to explain npm’s place in the ecosystem to VCs.“So what percentage of the fortune 500 use npm?”“100%”“Ha ha, but really”“No, really.”“What industries are you focused in?”“Every company that has a website.”“Isn’t that every company?”“Yes.”[326](#)

When faced with opportunities of this magnitude, the way to make meaning is to localize our interests. We can’t pay for the meals of every homeless person out there, but it’s easier to make the argument that we should occasionally buy lunch for the same homeless person whom we see every day. We might not care about the trash, air pollution, or safety of a city halfway across the world, but we do care about the safety and cleanliness of the streets that we walk on.

When it comes to vetting funding opportunities for open source, Ostrom’s principle of seeking “high context, low discount rate” opportunities serves us well. It’s not a company’s or individual’s job to fund *every* open source

project that they use. But some projects will mean more to certain funders than others, and those are the best opportunities to pursue.

It's OK (albeit mind-boggling) that a single developer might rely on thousands of open source projects to do their job every day, yet not personally fund the salaries of each one. But if there's a particular project that they happen to love, that's the one they should pursue. Likewise, it's the job of an open source developer to make themselves top-of-mind to a targeted set of funders instead of boiling the ocean. ("Everybody who uses my software should pay for my work.")

Critical to this argument is the idea of approaching funding from a place of abundance rather than scarcity. There will always be a loyal group of followers out there who are excited to support a given opportunity, so long as they have a meaningful relationship to it. This is especially true in today's world, where creators can attract millions of fans without being nationally or internationally famous. Whereas everyone once congregated around the same set of memes, now countless online celebrities are knighted every day.

In "1,000 True Fans," a blog post first published back in 2008 and now part of internet canon, the writer Kevin Kelly points to the value of a small, avid audience. In an updated version of this post, he writes,

To be a successful creator you don't need millions. You don't need millions of dollars or millions of customers, millions of clients or millions of fans. To make a living as a craftsperson, photographer, musician, designer, author, animator, app maker, entrepreneur, or inventor you need only thousands of true fans.

A true fan is defined as a fan that will buy anything you produce.^{[327](#)}

While we might debate the exact number of fans that creators ought to reach, the spirit of Kelly's post still stands. Even if an open source project is used by 100% of websites on the internet, the harder part, ironically, is making enough meaning for 1,000 people: becoming visible and present enough that they care what happens to you and your project, versus any other opportunity out there.

Tiago Forte, reflecting on his experience moving from public to paywall-protected content, observes,

The experience of blogging changed dramatically after I flipped the switch. My articles went from thousands of views to hundreds, but the quality of my readers spiked. I found my tribe. The noise of random passersby leaving inane comments dwindled to nothing, and we started having real conversations about what it would take to manifest a new vision of work. I started learning as much from them as they were learning from me. I went from having a blog that a large group of uncommitted readers perused, to a much smaller but more intimate group of people pre-committed to trying new things.^{[328](#)}

Who gets funded

The act of mixing money and open source frequently invites the following concern: “Won’t financial rewards adversely affect developers’ incentives to contribute?” Yes, but it depends on who you’re funding. §

Funding casual contributions is like paying people to leave comments on a creator’s work. It’s nonsensical, because these contributors are already

motivated to participate. What's more, maintainers are already swimming in casual contributions.

We wouldn't tackle the problem of too much email by making it *easier* for more people to send us email, or manage traffic congestion by paying *more* people to drive their cars across a bridge, instead of increasing the toll. In both cases, we'd want to *add* friction to the process, so that only the most important requests filter through. If anything, it's casual contributors who should be paying for access to maintainers, not the other way around.

Every casual contribution imposes a marginal cost to the maintainer, because they need to review and decide whether to merge it. This is why contributor initiatives like Hacktoberfest, despite addressing other issues (like making open source less intimidating to newcomers), don't solve for the ongoing costs of maintenance.

Hacktoberfest is an initiative that is sponsored by cloud infrastructure provider DigitalOcean and developer community DEV. (GitHub was a sponsor in previous years.) During the month of October, anyone who makes five pull requests to an open source project is eligible for a free T-shirt.³²⁹ This is a wonderful way to encourage newcomers to try their hand at making their first contribution. But it doesn't help to support the maintenance of open source projects, because casual contributions are already in abundance. Adding an extrinsic reward only encourages people to make spammy, low-quality contributions to claim it. (If you don't think this causes spam, trust me: it's incredible what developers will do for a free T-shirt.¶)

Similarly, bounty programs don't work well when they apply to all contributions, with payouts based on commits rather than on a contributor's

reputation. Paying for every contribution is not effective, because not every contribution needs to be paid for. As developer Ralf Gommers suggests, “Pay for things that require money (like a dev meeting) or don’t get done for free.”^{[331](#)}

Bounties can be effective when the tasks are vetted by maintainers themselves, and when they fund well-sscoped, finite tasks that are specialized or otherwise difficult to attract talent for, such as design work or database migration. For example, security bounties tend to work well because they tap into a wider pool of developers with specialized skills, encouraging them to tackle a one-off task that a project’s maintainers might not otherwise accomplish on their own. For security-related issues, it’s actually better when the participating developers *aren’t* already familiar with the codebase, because they bring a fresh set of eyes.

Crowdfunding campaigns can also work well because, like bounties, they fund bigger projects that require more focused time than contributions would normally allow. Font Awesome ran a Kickstarter campaign to fund the development of Font Awesome 5, a major update to their icon set, and raised just over \$1 million.^{[332](#)} And Linux kernel developer Kent Overstreet set up a Patreon to fund his work on bcachefs, “a next generation Linux filesystem.”^{[333](#)}

There is a lot of motivation, both intrinsic and extrinsic, that already powers open source work; we shouldn’t disturb the parts that are currently working. Instead, funding can become a useful motivator in places that are *absent* any other sort of motivation, such as the later stages of software maintenance. Because maintainers are more familiar with the project than

casual contributors are, and are responsible for most of the work done, it makes more sense to fund their time over anyone else's.

Eran Hammer, who maintains hapi.js, distinguishes between maintainer and contributor work on his Patreon page, which he uses to fund his time:

Maintaining hapi takes about 30-40 hours a month. This includes reading every issue in the core module and all the module it directly depends on, reviewing all pull requests and commits made to these modules, and answering complex questions. . . .

While the ecosystem as a whole is doing well, with a lot of community members contributing their time and resources, the core module is different. Decision [sic] made in core have an oversize impact on the community as well as put the health of the code and the application built on it at risk. . . .

Unless otherwise stated, these funds will not be shared with other hapi contributors. I don't want to create the impression that donating is sponsoring community work. The funds will go directly to me to cover my costs of writing open source software.³³⁵

Similarly, rather than fund new contributors, the Django Foundation raised money to create a Django Fellowship program.³³⁶ Tim Graham, the first fellow, was hired to manage the administrative aspects of Django, including patch review, issue triage, and release management. According to Loïc Bistuer, one of Django's core developers, "the churn rate on Django contributors has been very high historically and the Fellowship program is a direct answer to that."³³⁷ Having a Django Fellow ensured, for the first time, that pull requests were reviewed in a timely manner, and that releases stayed on schedule. In this case, paying for the work that Django

contributors weren't otherwise incentivized to do was hugely helpful to the project.

Funding people versus projects

Another question for open source funders is whether to fund the project itself or individual contributors. Funding the project builds institutional memory and makes it easier to manage funds transparently, but funding individuals provides greater flexibility and avoids the centralized governance issues that seem so antithetical to open source. Which is better?

Historically, funding was directed toward projects, and 501(c) foundations (that is, federally recognized nonprofits) were spun up to support the development of large, monolithic endeavors. Foundations could hold the trademark and IP assets of a project, and they also helped formalize corporate relationships to open source projects. “Funding individuals” mostly referred to companies hiring full-time developers to contribute toward a project.

Funding a project requires that developers have a legal entity to accept the funds—that is, unless funders are comfortable with paying an individual developer and hoping that person disburses those funds to others.

Otherwise, in order to accept funds as an organization, projects must either seek fiscal sponsorship from an umbrella organization, like the Software Freedom Conservancy or the Linux Foundation, or create a 501(c)(3) or 501(c)(6) foundation. They also need to define governance processes to manage who gets paid and what sort of expectations come with that money. Should every contributor get paid, or just the core developers? What if a core developer doesn't want to get paid? What about differences in geography, skill, and personal circumstances that might affect the payouts

among developers? What if someone is being paid but doesn't complete the work they promised?

Many developers don't want to manage this level of overhead, and as open source projects get smaller and lighter, it's especially hard to imagine them being managed by foundations. Does `left-pad`—the seventeen lines of code that right-justifies your text—need its own foundation? For precisely this reason, Open Collective, the aforementioned platform used by Vue.js to collect funds, offers a lightweight way to legally accept, manage, and disburse funds.

Today, as projects get smaller and a creator's reputation becomes increasingly prominent, funding individuals is becoming a more attractive option. Many open source projects are reluctant to start 501(c)(3) or 501(c)(6) nonprofit foundations, dissuaded by the bureaucracy that this step requires, compared to lighter-weight options available today.

From a governance perspective, funding individual developers is better aligned with the distributed nature of open source projects. To some developers, the fact that these projects aren't managed by formal organizations is a feature, not a bug. Funding individuals means that funding can come from multiple sources, that funding decisions are made based on the contributions of that particular individual, and that developers can join or leave without toppling the whole project.

Funding individuals can reduce some of the concerns that come with centralized project funding. One maintainer I spoke to explained that his co-maintainer lived in a country with a different standard of living. He didn't know how to distribute the funds they'd raised, because their salaries were so disparate. Another maintainer expressed frustration to me that the

project's author was drawing from their project's funds despite not actively contributing to the project anymore. He felt uncertain whether his reaction was warranted, given that the developer *was* the original author of the project, and probably deserved a reward.

When everybody is in charge of raising their own funds to support their own work, it reduces the amount of coordination required among the project's developers. Contributors are free to come and go as they please, and they have more flexibility in how they fund their time, whether it's with a full-time job, raising sponsorships, or working as a volunteer.

On the other hand, funding individuals can create a different set of governance challenges, and maintainers may find themselves effectively competing with one another for funding on the basis of their personal reputations. Maintainers might also disagree with another maintainer's decision to raise money, especially if it involves leveraging the project's brand to do so.

One Ruby project removed a maintainer who proposed launching a business that would offer paid support for the project. Although he had made significant contributions for years, the other maintainers didn't feel that gave him the right to commercialize the project itself, rather than finding ways to fund his own time.

Sometimes, users and contributors will bristle at a maintainer's decision to raise money. Matt Holt, for example, received negative feedback after introducing a commercial license to his project, Caddy, leading him to remark, somewhat bewilderedly, "We expected some pushback as usual, but the extreme controversy this created was unforeseen. Sorry about that."³³⁸ Oftentimes, however, a negative public reaction has more to do with

introducing changes to the user experience than with any philosophical opposition to open source developers making money.

Projects also tend to attract corporate funding better than individuals do, because companies are more comfortable with paying for code than for talent. In terms of value proposition, projects are attractive to corporate funders because they deliver commensurate benefits: code security and stability, influence, attracting hiring talent. And from an execution standpoint, funding projects is not unlike paying for software: they become a line item in a team’s budget, like expensing a SaaS (software as a service) product. By contrast, funding an individual developer is more like a contractor agreement, in which a company “hires” an individual for their services. These arrangements tend to be harder to pull off, both in terms of execution and making the case for the decision internally.

Raising money on the basis of personal reputation is liberating, in that developers like Sindre Sorhus aren’t tied to any one specific project. But it’s harder to rake in the big bucks without explicitly tying one’s work to a project. Sorhus himself notes,

My Patreon campaign is going well, but it’s mostly (amazing) individuals supporting me, which is not sustainable in the long-term. . . .

It’s a lot harder to attract company sponsors when you maintain a lot of projects of varying sizes instead of just one large popular project like Babel, even if many of those projects are the backbone of the Node.js ecosystem.³³⁹

Perhaps the biggest shift that comes from funding individuals is a cultural one. There's plenty of developer activity that takes place outside of what we typically think of as "the project," which is primarily driven by maintainers.

When GitHub launched its Sponsors product in 2019, the platform encouraged users to financially support the "developers, maintainers, writers, teachers, and programmers" that they depend on.³⁴⁰ Funding individuals opens up the possibility of sponsoring someone who makes great tutorials, writes books, answers support questions, or live-codes. It's less likely that these developers would directly benefit from money raised by an open source project, because they're just not as closely intertwined with "the project" as core developers are. Funding developers, rather than projects, changes what it means to fund open source.

^{*} Nota bene: ARPANET, the internet's predecessor, was commissioned by the US government. Governments funded the development of our earliest protocols, back when the internet was an R&D project. My skepticism about a government's ability to be a primary funder of open source software applies to digitally native public goods, such as today's system of projects built on top of those protocols.

[†] I also recommend Hickey's 2018 follow-up post, titled "Open Source Is Not About You," which can be found at <https://gist.github.com/richhickey/1563cddea1002958f96e7ba9519972d9>.

[‡] In a nod to the intrinsic motivation that seems to power some maintainers above other material concerns, when Stenberg announced he was leaving Mozilla without anything else lined up, and that Mozilla had effectively been cURL's primary sponsor, he added,

“Short-term at least, this move might increase my curl activities since I don’t have any new job yet and I need to fill my days with something . . .”

§ T. L. Taylor notes that similar concerns have plagued the world of gaming, where the professionalization and commercialization of games create a fear that play time has become “contaminated” or “corrupted.” Taylor argues that giving in to these fears would hamper the important work that livestreamers do today, and that work and play have always been culturally intertwined.³³⁰

¶ My favorite quote on this topic: “If the community offers its top 10 contributors a special T-shirt as an informative reward, the users may feel honored and appreciated even though the T-shirts have little value and therefore make more contributions in the future.”³³⁴

CONCLUSION

“Social media is in a pre-Newtonian moment, where we all understand that it works, but not how it works.”

—**KEVIN SYSTROM**, cofounder, Instagram³⁴¹

Our online social spaces are littered with the artifacts of our creative endeavors. Today, “content” is better understood not as a thing we set out to make—as an automaker might exist solely to produce cars—but, as a friend wrote in an email to me, as “an externality from [our] existing social systems.”³⁴² Content is a snapshot of our civilization.

The history of software gives us insight into how our attitudes toward content are changing. In the twentieth century, code was bundled into physical formats—a book, a floppy disk, a CD—which made it easier to price and sell. As code became liberated from these formats, and eventually distributed under open source licenses, it became harder to directly charge for. With millions of lines of code freely available today, the focus has shifted from *what* developers make to *who* they are.³⁴³

Python developer Shauna Gordon-McKeon once posed a hypothetical question to me: “Take a platform you love. Would you rather lose access to all the past content your connections have posted, or lose the connections themselves?”³⁴⁴ Her point was that the value created on these platforms doesn’t lie in the content itself so much as in the underlying social graph. Our relationship to content matters less than our relationships to the people who make it. As a result, we’re starting to treat content not as a private economic good but as the externalization of our social infrastructure.

Platforms have helped bring about this shift more quickly. By reducing the costs of production and distribution, they’ve made it easier for creators to

function as one-man operations. *Stratechery*'s Ben Thompson calls this the “faceless publisher” model, wherein “atomized content creators, fueled by social media, build their own brands and develop their own audiences; the publisher, meanwhile, builds scale on the backside, across infrastructure, monetization, and even human-resource type functions.”³⁴⁵

What's more, by bringing creators themselves to one place, platforms have turned content production into a status game. Eugene Wei calls these “status as a service (StaaS)” businesses, which flourished after Facebook's invention of the News Feed, and other platforms' launches of their own social feeds, “unleashed a gold rush for social capital accumulation.”³⁴⁶

How, then, should we think about the production of content today, in light of platform-creator relationships? How does the “atomization” of production affect prior theories about how people produce, whether Coase's theory of the firm, Ostrom's common pool resources, or Benkler's peer production?

By treating content as a commodity, we risk solving the wrong puzzle. Finding answers means returning to the essential questions. Chapters 2 and 3 looked at the social dynamics between creators and the communities that form around them. I suggested that one-to-many models, typical among online creators, are *centralized communities*, with hidden roles played by both platforms and the creator's audience; these communities stand in contrast to the distributed, many-to-many online communities we're used to.

Chapter 4 revisited the question of marginal cost. We tend to assume that content doesn't incur significant marginal costs, thanks to platforms that

now absorb most of the distribution costs for creators. However, it's the *maintenance* of content that incurs hidden costs with time and use.

A creator's *reputation* also requires a certain type of maintenance. Reputation has a half-life on any social platform; successful creators accumulate reputation, which serves as a "battery" that helps them store consumer attention. But if they don't keep producing new work, that battery will degrade, and eventually get depleted.

Finally, in Chapter 5 I suggested that there are two economic goods in open source masquerading as one. Open source code is *consumed* like a public good, because when code is viewed as a static commodity the cost of additional users is nearly zero. But open source code is *produced* like a commons, where a maintainer's attention is the limited resource.

Maintainers must be mindful of how they allocate their attention, avoiding extractive contributors.

"Public" does not imply "participatory"; it's possible to freely distribute content without depleting producer attention. The remainder of Chapter 5 examined various strategies that developers employ to manage their attention, whether it's relying upon automation, limiting their availability, or distributing costs onto users.

Historically, most of our questions about the value of content have focused on the distribution side, rather than the production side. Today, the most interesting questions we can ask will focus on *how* content is made and maintained, and *by whom*. We've previously treated content as a first-copy cost problem, and have developed solutions like patents, intellectual property, and copyright to incentivize its creation. But these solutions don't address the costs of maintenance, which accrue over time. The challenges

facing online creators today derive from the fact that they are playing a repeated game, not a single one.

Maintenance costs are partly caused by poorly defined boundaries around online participation, which haven't scaled to the way we interact with one another today. Creators face a problem of over-participation, not under-participation, in managing expectations from their audience.

Another type of maintenance cost comes from the accumulation of reputation. Every platform that creates social capital for creators must also account for the cost of maintenance, because a creator's reputation degrades over time, unless they continue to invest in it.

I've spent most of this book examining this behavior in open source, which helped me understand the problems of scale that affect our online world more broadly. I'll spend the final pages zooming out in order to explore how what we've learned can be applied not just to open source developers but other online creators. I'll focus on two areas in particular: the problem of managing over-participation and the problem of making money.

MANAGING OVER-PARTICIPATION

When people talk about the “attention economy,” they’re usually referring to the *consumer’s* limited attention, as when multiple apps compete for a user’s time. But a *producer’s* limited attention is just as important to consider.

As audience engagement grows, more consumers appropriate attention from creators. So it’s critical to find ways to manage this demand, especially extractive demand, which includes comments, reactions, direct messages,

and other requests for interaction. A tragedy of the commons occurs not from consumers over-appropriating the content itself, but from consumers over-appropriating a creator's attention.*

Why is this problem so hard to solve for? Part of it comes from lingering social norms around the expectations of unfettered participation in our online social spaces. Another problem is that we currently lack the infrastructure to manage this demand. As social norms change, our platforms are evolving to address them. Like a bridge that needs to support more traffic than it was built for, every social platform is scrambling to upgrade its infrastructure to accommodate the volume of social interactions we're dealing with today. Platforms need to build skyscrapers where there were once villages.

Our social platforms were built for distributed, small-scale, many-to-many use cases: the quaint social world of yesteryear. They were modeled after internet forums, chat groups, and mailing lists, because these were the only blueprints that we had for our online social infrastructure.

As more people flocked to these platforms, we saw a few bridges collapse, because, it turns out, we had no idea how to build bridges for today's world. The aftermath of the 2016 United States presidential election saw Facebook wholly unprepared for its new role as a steward of civil society. But platforms are learning from those mistakes and rebuilding their services with new knowledge that we're still uncovering today.

If you imagine a music concert that keeps adding attendees until it becomes a rousing crowd, you can understand the effect of adding more people to our social platforms. When six people are gathered, they can all talk to one another; everyone is engaged in the same conversation. When a thousand

people are gathered, they divide into two types of interaction. There's the *broadcasting* effect, when someone climbs onstage to control the crowd and everyone turns to watch. And then there's the *small-group* effect, when people strike up side conversations with their neighbors, ignoring the main stage.

Social platforms must rebuild their infrastructure to accommodate these two use cases. Yancey Strickler, who cofounded Kickstarter, calls this the “dark forest theory” of the internet: “an increasing number of the population has scurried into their dark forests” to avoid the mainstream web, which has become “a relentless competition for power.”³⁴⁷ He points to newsletters, podcasts, and group chats as examples of dark forests, while Facebook and Twitter are examples of the mainstream, which will continue to exist alongside more private channels. Mark Zuckerberg himself declared to a crowd at Facebook’s 2019 F8 conference that “the future is private,” insisting, “Over time, I believe that a private social platform will be even more important to our lives than our digital town squares.”³⁴⁸

The public stage increasingly reflects a *one-way mirror* pattern, where anybody can consume content but interactions with its creator are limited. Twitter has begun to experiment with giving its creators more moderation control, such as hiding replies to tweets and limiting who can reply to tweets.³⁴⁹ The Instagram Stories feature (a format initially invented by Snapchat) is designed for broadcasting rather than conversing with one’s audience. Kevin Systrom, Instagram’s CEO when Stories launched, noted at the time that one reason for introducing Stories was to give creators a place to share without the pressure of likes and comments.³⁵⁰ Podcasts are surging in popularity, with nearly one in three Americans saying that they listen to podcasts at least once per month—a number that doubled from

2014 to 2019.³⁵¹ And newsletters are another, curiously reemerging format, where creators can broadcast in longform to large audiences, without expectation of a dialogue.

In the private sphere, small group chats are having a moment on messaging apps like Facebook Messenger, Instagram, iMessage, WhatsApp, Signal, and Telegram. Group chats are the embodiment of Ostrom's thesis: a way of drawing boundaries around one's community, excluding extractive contributors, in order to allow for higher-context interactions. They, too, are an adaptive measure to the problem of over-participation.[†]

Finally, there are still distributed, many-to-many social patterns, such as on Reddit and Facebook groups, that are reminiscent of old online forums. These still work in limited contexts, according to the conditions identified by Benkler. When social interactions are modular and granular, with fungible membership and low coordination costs, they can scale costlessly. Blogger Kicks Condor, in an email to me, describes Reddit as "this massive N:N conversation that almost starts to feel like a singular voice because the names and identities are so muted."³⁵² As a result, these communities are frequently oriented around specific interests, whether it's a local moms' Facebook group or a subreddit about dating and relationships. Like small group chats, they are less dependent upon a specific platform, compared to creators who broadcast in one-to-many formats.

Fundamentally, social platforms like Twitter and Facebook still follow the post-and-comment model of an earlier era, where interactions with the author are frictionless, even encouraged. These platforms struggle, more than any other, to adapt to modern social needs.

One problem is that these platforms assume that all users are interchangeable, whereas in a one-to-many broadcasting format creators are a particular, non-fungible type of user. The experience of an Instagram user with 2 million followers is different from that of an Instagram user with 200 followers. The former is the epicenter of a centralized community, whereas the latter is either passively consuming 1:N content or participating in a small group. Instagram has started to recognize these differences with designated “creator accounts,” which allow their creators to better manage their inbound requests using multiple inboxes, hidden contact information, and automated replies.³⁵³

While creators are the focus of attention, they don’t do it all alone. There are a number of supporting roles that enable them to do their work, whether it’s fans who serve as “moderators” and weed out bad actors or those who curate a creator’s work to make it more digestible to others. As discussed in previous chapters, the 1:N social pattern is itself a community, even though it doesn’t fit our typical definition of an online community. Therefore, we need to take a holistic approach to supporting creators, enabling dedicated supporters to pitch in, instead of treating the creator as a single unit. They’re a tree in a forest, not a potted plant.

Designing for one-to-many interactions isn’t an entirely new challenge—these types of creators have spiritual roots in television or radio broadcasting—but the *social* aspect makes things murky. By sharing so much of themselves, these creators occupy a strange space between the intimate and the pure broadcaster experience.

The platform-creator relationship makes things even more confusing, because it’s not always clear whether a creator is operating in the platform’s

space or in their own. Moderation, for example, requires that communities be empowered to self-organize and manage their own rules, as we discovered from Ostrom's work. Platforms ought to give creators the tools they need to manage their own communities. But platforms also face enormous public pressure to develop their own moderation policies, which are difficult to apply indiscriminately to every creator and their respective community.

Many platforms, including Instagram and YouTube, are experimenting with hiding like counts and defaulting to hidden comments, in order to reduce demand from extractive contributors.³⁵⁴ But Instagram's experiment hides like counts from the public—not from creators themselves. This approach runs the danger of destroying the reputation economy they've created. Early on, aspiring creators who are building an audience want—even need—these social signals, and this isn't necessarily a bad thing. These reputational systems help us measure and assign value. It's only later on, with the costs that come with maintaining one's reputation, that a high volume of likes and comments can become exhausting.

The ability to hide, mute, or turn off these interactions is important to creators at later stages. Taylor Swift explains how she maintains her sanity while occupying a highly visible position:

One thing I do to lessen this weird insecurity laser beam is to turn off comments. Yes, I keep comments off on my posts. That way, I'm showing my friends and fans updates on my life, but I'm training my

brain to not need the validation of someone telling me that I look .

I'm also blocking out anyone who might feel the need to tell me to "go die in a hole ho" while I'm having my coffee at nine in the morning.³⁵⁵

Another helpful approach is to encourage the role of curators, who can serve as highly complementary and beneficial counterparts to creators, as we saw in the example of Twitch streamers working with their moderators, or Rich Hickey, the author of Clojure, working with Alex Miller, a Clojure maintainer. Curators help moderate the public space for a creator, filtering out extractive requests and only surfacing the things that most require a creator's attention. As we saw in Chapter 5, creators and platforms can't manage all their moderation needs manually, so they need to lean heavily upon user-to-user solutions.

The writer Tavi Gevinson, who grew up in the public eye, explains how she manages her Instagram presence with the help of a curator:

I asked a woman who had done personal-assistant work for me if she wanted a new gig. Since then, I've texted her my photos and captions, and she has posted them on my behalf. . . .

. . . . As far as other people's posts go, I still look at some accounts on my computer as though they were blogs. Sometimes I check comments there, too, where it's less tempting to get sucked in, and this year, I started a Google doc where this person pastes any feedback I might be interested in, according to the criteria I've specified: the kind of personal, detailed comments I used to see on Rookie or my blog; constructive criticism; and anything from a verified account.^{[356](#)}

Finally, we might think of the “like,” and its subsequent permutations, as a form of automation. By making social interactions more lightweight, platforms can reduce the amount of work that creators need to do.

The “like” was an adaptive mechanism to reduce the attention cost of comments. While it reduces the friction for anyone to participate, it also reduces a creator’s mental overhead, similarly to how continuous integration and testing have reduced the amount of review that maintainers need to do for each pull request. (Relatedly: one of the features requested by GitHub’s open source maintainers in their “Dear GitHub” letter was a voting mechanism that would replace the overwhelming volume of “+1” comments they received, “which serve only to spam the maintainers and any others subscribed to the issue.”³⁵⁷ GitHub added emoji reactions in 2016, which function similarly to likes, and helped reduce comment overhead.³⁵⁸)

Emoji, which took off in the 2010s, are another example of social micro-interactions that helped meet growing demand for people’s attention. It’s easier to send an emoji to convey a certain emotion than to type out full sentences. Likes and emojis fit Benkler’s condition of *granularity*, which makes it possible for distributed social interactions to scale. Apple has added tapbacks to iMessage, and Facebook Messenger has added reactions to group chats, both of which make it easier to acknowledge and respond to messages.

When it comes to internet fame, we have no notion of “organization building” to support high-demand creators as their audiences grow. Nor do creators always have the financial resources to manage growing demand on their own, since online fame doesn’t always translate into money. Big-name celebrities and CEOs eventually hire security details and administrative support. Similarly, platforms must provide the support systems that creators need to be able to thrive.

While we, as users, can try to devise our own solutions, we're often limited by what platforms allow us to implement. Platforms provide the infrastructure for our online social spaces, and their design decisions massively shape our daily experiences. To borrow from anthropologist Edward T. Hall, who wrote about the social influence of "fixed-feature" spaces like buildings: platforms are "the mold into which a great deal of behavior is cast."³⁵⁹

MAKING MONEY

If the one-way mirror—where anybody can consume but only some people participate—is an adaptive response to over-participation, it’s unsurprising that these smaller, focused communities are giving rise to a particular set of models for funding creative work.³⁶⁰

The first wave of monetization was about making meaning of the big. Display advertising, for example, operates as a function of viewership. Pricing is typically based on CPM, or cost per thousand impressions, and CPC, or cost per click, both of which require a lot of views to make these numbers meaningful. These days, as consumers are faced with near-infinite options, making money as a creator is about making meaning of the small again: Kevin Kelly’s “1,000 true fans,” finally come to pass. We’re seeing renewed interest in subscription models, sponsorships, and merchandise, all of which operate as a function of parasocial, or one-sided, intimacy.

It’s hard to write about the economics of content without discussing advertising further, but because it’s such a big topic on its own, I’m not explicitly covering it in this book. Suffice it to say that the rise of subscription models shouldn’t be read as a death knell for advertising. These two models aren’t mutually exclusive; advertising is still alive and well, and it will continue to work for a very long time. And product sponsorships will probably only become more valuable, because they’re driven by the strength of a creator’s reputation. The writer Tim Ferriss, for one, found that advertising worked better than subscriptions for his podcast, because his listeners especially trust his product recommendations.³⁶⁰

As for subscriptions, paywalls might seem like a way to monetize *content*, but, really, what creators are monetizing is their *community*, whether by offering a closer connection to the creator, like-minded people to talk to, or a safe haven from extractive contributors.

A paywall is more like the ticket kiosk at a theme park than a price tag on a car. Most comments aren’t useful, so charging readers to leave a comment helps ensure that participants have meaningful skin in the game. Creators charge money in order to make social interactions meaningful again.

Something Awful, an internet forum that was especially popular in the mid-2000s, charged its users an “activation fee.” While anybody could read the site’s forums (minus an occasional sign-up banner blocking their view), only paying members could post. Kevin Bowen, a former administrator, explained in a *Vice* interview that “when [Richard Kyanka, who created Something Awful] started charging for forums accounts registrations, he wasn’t doing that to make money. He was doing that because he was sick of banning people from the forums.”³⁶¹

Jon Hendren, another former Something Awful administrator, chimed in, “When Rich put the paywall in effect, it kept idiots out to an enormous degree. . . . You have to put in a little investment if you want to participate.”

With this principle in mind, it’s easier to see how paywalls are frequently misapplied. Putting all content behind a paywall isn’t likely to work well for creators or publishers, with the exception of a few heavy hitters (say, *The New York Times*). Similarly, micropayments—where consumers pay small sums for a specific piece of content, such as paying to unlock a specific article—don’t usually add up to much either (again, unless you’re *The New York Times*, and can play upon a reader’s fear of missing out).

Micropayments make the transaction about content, rather than about creators, but because there is so much freely available, highly substitutable content they create decision fatigue for consumers.^{[362](#)}

Subscription models can operate like a freemium model, but they get even more interesting as a two-sided market. In a freemium model, a creator gives away some of their content for free, but restricts other content to those with paid subscriptions. The free content helps creators grow their reputation via public network effects.

In a two-sided market, paying subscribers subsidize all of the content for nonpaying readers, under the assumption that creators aren't actually selling content but a sense of membership and identity. Instead of charging, say, all 100,000 readers ten cents to read an article, creators can instead give away the article for free, but charge 1,000 extra-dedicated subscribers ten dollars per year. Journalist Tim Carmody, for example, subsidizes his newsletter *Amazon Chronicles* this way.^{[363](#)} Paying subscribers make it possible for him to offer the newsletter for free to everyone:

The most powerful and interesting media model will remain raising money from members who don't just permit but insist that the product be given away for free. The value comes not just what [*sic*] they're buying, but who they're buying it from and who gets to enjoy it.

The bigger those two pools get—the bigger the membership, and the bigger the audience—the better it gets for everyone. . . . PBS as a service.^{[364](#)}

Matthew Butterick, who wrote the popular book *Practical Typography*, also found success by targeting a subset of his readers. Matthew offers his book for free online, but he didn't want to plaster it with “nonstop wheedling and

begging” for money, as “that would’ve made the book permanently ugly & annoying.”³⁶⁵ After noticing that much of his unpaid traffic came from just fifteen to twenty websites, he decided to target only visitors from those websites, suggesting that they pay for the book. As a result, the number of direct payments doubled that year, which helped him continue to offer the book for free to more than 600,000 readers annually.

Finally, the news industry is a useful case study to demonstrate how different funding models might apply to different types of content creators. Like “open source,” “news” doesn’t refer to just one type of content. Also like open source, there are a few big newspapers (*The Washington Post*, *The Wall Street Journal*, *USA Today*, and others), which don’t fully capture the experience of the rest of the news industry, much like Linux doesn’t reflect the experience of most open source projects. With the continued unbundling of newspapers, we’ll likely see different models that work for different types of writers. As Damon Kiesow, a professor at the Missouri School of Journalism, puts it, “National and local journalism are not the same business. The New York Times can chase scale for profit, The Centre Daily Times can’t. Because what the New York Times has is an audience. In College Park, they have a community.”³⁶⁶

Social media is arguably supplanting the need for “breaking news” reporters. Like casual contributors to an open source project, eyewitnesses post about a breaking news event because they are intrinsically motivated to do so. The fact that local news outlets frequently appear in eyewitnesses’ comments, asking them for the right to use their images, tells us that average citizens are becoming a better breaking news source for major media outlets, rather than the other way around.

Breaking news is the equivalent of casual contributions in open source: those who are closest to the problem have an intrinsic desire to “contribute” with no expectation of financial reward, but also have no desire to stick around after making their contribution. Most people who post about a breaking news event don’t want to become news reporters; they just want to share and move on. Similarly, most casual contributors have no desire to join a “contributor community” or become maintainers; they’re there to fix their own problem and move on.

For news reporting that focuses on a specific vertical, we’ll see a shift that looks like the shift from “funding projects” to “funding people,” which we explored in Chapter 5. As media companies shift toward “atomized” creators, journalists will find it easier to independently monetize their own reputations, much like any other content creator. Readers will pay for a curated, high-quality take on their interests, as well as, sometimes, a sense of community with other like-minded readers.

For example, former journalist Azeem Azhar brings in six figures of annual recurring revenue with his technology newsletter, *Exponential View*.³⁶⁷ *The Athletic* built its subscription platform, focused on sports, by recruiting high-reputation sports writers from local newspapers.³⁶⁸ Even reporting on the local news is a form of “niche journalism,” as in the case of Darrell Todd Maurina, who single-handedly curates the local news for Pulaski County, Missouri, on his Facebook page.³⁶⁹

Overall, I’d expect to see in the news industry something similar to what happened in open source: a “dumbbell”-shaped distribution of contributors. On one end are purely casual contributors (breaking news, casual punditry, and commentary on the news), who post on social media because it’s low-

effort with the potential for a bit of upside, and no expectation of maintenance, funding, or having to make ongoing contributions. On the other end are the maintainers (news columns, investigative journalism, and features): those with in-depth knowledge of their topic, who make highly non-fungible contributions, and who will make money off their reputations.

For those in the latter category, the name of the game is building highly targeted audiences and optimizing for fewer people paying more money, rather than shilling for page views. Podcasts, newsletters, and longform writing are attractive these days because they help creators filter for niche audiences, rather than losing potential value to broad reach on public channels. Instead of maximizing for likes, these creators maximize for meaning.

An interesting implication here is that a creator's relevance (to *some* niche audience) matters more than quality or trust on a global scale. Early newspaper and media brands built their reputations on the promise of truth and "objectivity," whereas it seems more likely that future media brands will build their reputations on the promise of relevance.

I'd argue that the reason this funding shift has taken so much longer in journalism—much like in open source—compared to other forms of content creation is because it requires sloughing off preexisting social norms. Twitch and YouTube creators, for example, essentially created their medium along with the platform, whereas open source and journalism are still grappling with the inherited models of their predecessors.

Transitions are hard, but when it comes to making money as a creator online, I feel more optimistic about these opportunities than ever. We're moving toward a future where rewards are heavily influenced by the quality

of one's audience more than its size. This affords creators an enormous degree of freedom and helps perpetuate the renaissance of ideas that is already well underway. We don't have all the answers yet, but I'm hoping this book helps point us toward the right questions.

^{*} Another way of describing this pattern is through *negative cross-side network effects*. Cross-side network effects describe how value is created between two groups of users: in this case, creators and their users. Ideally, the network creates *positive* cross-side effects, where the presence of more creators and users is mutually beneficial. But too many users can also create a *negative* cross-side effect, which reduces value to the creator.

[†] Even open source projects have a version of this, such as private groups for maintainers to discuss sensitive topics (like security issues) or new features that they're not yet ready to announce.

[‡] If we're looking at the world of content, there's plenty to explore in the realm of subscription-based services, like Netflix, Spotify, Apple News, or Disney+, but I'll leave that analysis to others. This book is primarily focused on the people *behind* the content, so I'm limiting my focus to the economies that arise on social platforms like GitHub, Instagram, Twitter, or Twitch, and the creators who build their reputations on them.

CREDITS

This book would not exist without Josh Greenberg, who encouraged me to take this project seriously and helped me refine a jumbled set of ideas into something I could stand to look at. I wouldn't have made it through an initial draft of this book without our regular check-ins and Josh's cheerful optimism. Thank you for your support.

Thank you to Juan Benet for giving me a home at Protocol Labs, where I had the creative space to research and write this book, and to Evan Miyazono for protecting my time and believing in the value of this project. Thank you also to Harvard University for giving me access to their library and resources over the course of my research. All of these arrangements made it possible for me to do this work the way I wanted, as an independent researcher. I feel very lucky.

Thank you to Patrick Collison, Brianna Wolfson, Sid Orlando, and Kate Lee for welcoming me into the Stripe Press family and being so wonderful to work with. I couldn't imagine a better home for this book. Thank you to Alex Padalka and Susannah Kemple for whipping this book into shape.

Thank you to Titus Brown, John Backus, Michael Nielsen, Philip Guo, Devon Zuegel, and Liz Voeller for pointing me to new ideas that majorly influenced the direction of this book. Thank you to my coworkers at GitHub, whom I learned so much from, and who made it such a magical place to work. Thank you to Mike McQuaid, Henry Zhu, Feross Aboukhadijeh, Nat Friedman, Zooko Wilcox, and all the open source developers who've taken time to share their experiences with me and provide feedback over the years. I'm grateful to have spent my time with such an incredibly thoughtful, creative, and generous group of people.

Thank you to every developer who's tweeted, written a blog post, given a conference talk, posted to a forum or mailing list, opened an issue or pull request, or otherwise shared something of themselves in public. I wanted this book to showcase the real stories of developers, and I greatly benefited from being able to dig through the vast repository of information out there. Thank you also to everyone who replied to my tweets, newsletters, and blog posts throughout the writing process: you gave me a sounding board to test and refine many of the ideas that eventually made their way into this book.

NOTES

INTRODUCTION

- [1](https://www.w3.org/History/1989/proposal.html) Tim Berners-Lee, “Information Management: A Proposal,” The Original Proposal of the WWW, HTMLized, May 1990,
<https://www.w3.org/History/1989/proposal.html>.
- [2](https://www.fordfoundation.org/about/library/reports-and-studies/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure) Nadia Eghbal, “Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure,” Ford Foundation, July 14, 2016,
<https://www.fordfoundation.org/about/library/reports-and-studies/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure>.
- [3](https://doi.org/10.1109/saner.2016.68) Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa, “More Common Than You Think: An In-Depth Study of Casual Contributors,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Suita, Japan: IEEE, March 2016): 518–528,
<https://doi.org/10.1109/saner.2016.68>.
- [4](https://trends.builtwith.com/docinfo/Twitter-Bootstrap) Twitter Bootstrap Usage Statistics,” Built With, accessed March 31, 2020, <https://trends.builtwith.com/docinfo/Twitter-Bootstrap>.
- [5](https://github.com/twbs/bootstrap/graphs/contributors) “Contributors: Commits,” Bootstrap Insights, GitHub, accessed March 31, 2020,
<https://github.com/twbs/bootstrap/graphs/contributors>.
- [6](https://github.com/nayafia/user-support/blob/master/top-100-by-issue-volume.csv) Nadia Eghbal, “User Support System Analysis,” Nayafia Code, Github, September 27, 2018, <https://github.com/nayafia/user-support/blob/master/top-100-by-issue-volume.csv>.

7 Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies, “Why Software Projects Need Heroes (Lessons Learned from 1000+ Projects),” ArXiv, April 22, 2019, <https://arxiv.org/pdf/1904.09954.pdf>.

8 Npm, Inc., “This Year in JavaScript: 2018 in Review and Npm’s Predictions for 2019,” *Npm* (blog), Medium, December 6, 2018, <https://medium.com/npm-inc/this-year-in-javascript-2018-in-review-and-npms-predictions-for-2019-3a3d7e5298ef>.

9 Steve Weber, *The Success of Open Source* (Cambridge, MA: Harvard University Press, 2005), Loc 867.

10 “The State of the Octoverse,” GitHub, 2019, <https://octoverse.github.com/>.

11 DinoInNameOnly, “Most of What You Read on the Internet Is Written by Insane People,” R/slatestarcodex, Reddit, October 27, 2018, https://www.reddit.com/r/slatestarcodex/comments/9rvroo/most_of_what_you_read_on_the_internet_is_written/.

12 CBS News, “Meet the Man behind a Third of What’s on Wikipedia,” CBS News, CBS Interactive, January 26, 2019, <https://www.cbsnews.com/news/meet-the-man-behind-a-third-of-whats-on-wikipedia/>.

13 Kristen Roupenian, “What It Felt Like When ‘Cat Person’ Went Viral,” *The New Yorker*, January 9, 2019,

<https://www.newyorker.com/books/page-turner/what-it-felt-like-when-cat-person-went-viral>.

01

[14](#) “The State of the Octoverse,” GitHub, 2019,
<https://octoverse.github.com/>.

[15](#) Free Software Foundation, “What Is Free Software?,” GNU
Operating System, July 30, 2019,
<https://www.gnu.org/philosophy/free-sw.en.html>.

[16](#) Nicole Martinelli, “Walking the Walk: Why It’s a Crooked Path for
Free Software Activists,” Super User, February 8, 2019,
<https://superuser.openstack.org/articles/walking-the-walk-why-its-a-crooked-path-for-free-software-activists/>.

[17](#) Steven Levy, *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition* (Sebastopol, CA: O’Reilly, 2010).

[18](#) Linus Torvalds (torvalds), “Add Support for AR5BBU22 [0489:e03c],” Linux Pull Requests, GitHub, May 11, 2012,
<https://github.com/torvalds/linux/pull/17#issuecomment-5654674>.

[19](#) Eric S. Raymond, “Sex Tips for Geeks,” Catb, n.d.,
<http://www.catb.org/esr/writings/sextips/>.

[20](#) Eric S. Raymond, “Eric’s Gun Nut Page,” Catb, March 19, 2015,
<http://www.catb.org/~esr/guns/>.

21 Aalto University Center for Entrepreneurship (ACE), “Aalto Talk with Linus Torvalds [Full-Length],” June 15, 2012, YouTube video, 49:58, <https://www.youtube.com/watch?v=MShbP3OpASA>.

22 Richard Stallman, “Why Open Source Misses the Point of Free Software,” GNU Operating System, January 7, 2020, <https://www.gnu.org/philosophy/open-source-misses-the-point.html.en>.

23 Eric S. Raymond, “Project Structures and Ownership,” *Homesteading the Noosphere*, Catb, August 24, 2000, <http://catb.org/~esr/writings/homesteading/homesteading/ar01s16.html>

.

24 Dawn Foster, “Who Contributes to the Linux Kernel?,” The New Stack, January 18, 2017, <https://thenewstack.io/contributes-linux-kernel/>.

25 “How the Development Process Works,” The Linux Kernel, accessed April 15, 2020, https://www.kernel.org/doc/html/latest/process/2_Process.html.

26 Nadia Eghbal, “There Is No ‘My’ in Open Source,” Medium, March 24, 2016, <https://medium.com/@nayafia/there-is-no-my-in-open-source-c3e5555390fa>.

27 “About GitLab,” GitLab, accessed March 31, 2020, <https://about.gitlab.com/company/>.

28 “The State of the Octoverse,” GitHub.

29 Eric Wong, “Re: Please Move to Github,” Unicorn Ruby/Rack Server User+Dev Discussion/Patches/Pulls/Bugs/Help, August 1, 2014, <http://bogomips.org/unicorn-public/20140801213202.GA2729@dcvr.yhbt.net/>.

30 Stack Overflow Insights, “Developer Survey Results 2018,” Stack Overflow, 2018,
<https://insights.stackoverflow.com/survey/2018#work-version-control>.

31 “The State of the Octoverse,” GitHub.

32 Ben Balter, “Open Source License Usage on GitHub.com,” *The GitHub Blog*, GitHub, March 9, 2015, <https://github.blog/2015-03-09-open-source-license-usage-on-github-com/>

33 Brett Cannon, “The History behind the Decision to Move Python to GitHub,” *Tall, Snarky Canadian*, January 13, 2016,
<https://snarky.ca/the-history-behind-the-decision-to-move-python-to-github/>.

34 *App: The Human Story*, directed by Jake Schumacher, 2017,
<http://appdocumentary.com/>.

35 Monika Bauerlein and Clara Jeffery, “How Facebook Screwed Us All,” *Mother Jones*, March/April 2019,
<https://www.motherjones.com/politics/2019/02/how-facebook-screwed-us-all/>.

36 Kurt Wagner, “Facebook’s Acquisition of Instagram Was the Greatest Regulatory Failure of the Past Decade, Says Stratechery’s

Ben Thompson,” *Vox*, June 2, 2018,
<https://www.vox.com/2018/6/2/17413786/ben-thompson-facebook-google-aggregator-platform-code-conference-2018>.

37 Cannon, “The History behind the Decision to Move Python to GitHub.”

38 The Carters, “Nice,” *Everything is Love*, 2018,
<https://genius.com/The-carters-nice-lyrics>.

39 Jacob Thornton, “What Is Open Source & Why Do I Feel So Guilty?,” Dotconferences, November 30, 2012, Youtube video,
<https://www.youtube.com/watch?v=UIDb6VBO9os>.

40 Steve Klabnik, “What Comes after ‘Open Source,’” *Steve Klabnik* (blog), April 2, 2019, <https://words.steveklabnik.com/what-comes-after-open-source>.

41 Mikeal Rogers, “The GitHub Revolution: Why We’re All in Open Source Now,” *Wired*, March 7, 2013,
<https://www.wired.com/2013/03/github/>.

42 Russ Cox, “Our Software Dependency Problem,” *Research!rsc*, January 23, 2019, <https://research.swtch.com/deps>.

43 “The State of the Octoverse,” GitHub.

44 Stack Overflow Insights, “Developer Survey Results 2018.”

45 Kent C. Dodds, “Big Announcement: I’m a Full-Time Educator!,” *Kent C. Dodds* (blog), February 17, 2019,

<https://kentcdodds.com/blog/full-time-educator>.

46 Antoni Kepinski, Jerod Santo, Feross Aboukhadijeh, and Mikeal Rogers, “Building PizzaQL at the Age of 16,” *JS Party*, podcast audio, July 26, 2019, <https://changelog.com/jsparty/85>.

47 Dan Abramov, “Things I Don’t Know as of 2018,” *Overreacted*, December 28, 2018, <https://overreacted.io/things-i-dont-know-as-of-2018/>.

48 Henry Zhu, “In Pursuit of Open Source (Part 1),” *Henry’s Zoo*, March 2, 2018, <https://www.henryzoo.com/in-pursuit-of-open-source-part-1/>.

49 Kent C. Dodds and Sarah Drasner, “An Open Source Etiquette Guidebook,” CSS-Tricks, December 8, 2017, <https://css-tricks.com/open-source-etiquette-guidebook/>.

50 “Sindre Sorhus Is Creating Open Source Software,” Patreon, accessed March 31, 2020, <https://www.patreon.com/sindresorhus>.

51 “Feross Is Creating Open Source Software like WebTorrent and Standard,” Patreon, accessed March 31, 2020, <https://www.patreon.com/feross/>.

52 Dan Abramov (@Dan_Abramov), “I’m Sorry for Disappearing (...),” Twitter, August 27, 2019, 8:55 a.m., https://twitter.com/dan_abramov/status/1166333416272486400.

53 Linus Torvalds, “Linux 4.19-rc4 Released, an Apology, and a Maintainership Note,” LKML, September 16, 2018,

[https://lkml.org/lkml/2018/9/16/167.](https://lkml.org/lkml/2018/9/16/167)

54 Pamela Chestek, “Member conduct,” [License-discuss], February 28, 2020, https://lists.opensource.org/pipermail/license-discuss_lists.opensource.org/2020-February/021350.html.

Eric S. Raymond, “The Right to Be Rude,” *Armed and Dangerous*, February 27, 2020, <http://esr.ibiblio.org/?p=8609>.

55 Richard Stallman, “Political Notes from 2019: July - October,” Richard Stallman’s Personal Site, October 31, 2019, <https://stallman.org/archives/2019-jul-oct.html>.

Free Software Foundation, “Richard M. Stallman Resigns,” Free Software Foundation, n.d., <https://www.fsf.org/news/richard-m-stallman-resigns>.

56 Sindre Sorhus (@sindresorhus), “An observation after having . . .,” Twitter, December 8, 2016, 2:03 p.m., <https://twitter.com/sindresorhus/status/806937150575017984>.

57 Sindre Sorhus (@sindresorhus), “Some observations from having . . .,” Twitter, May 21, 2019, 7:03 a.m., <https://twitter.com/sindresorhus/status/1130791267393163267?s=21>.

58 Sindre Sorhus (@sindresorhus), “I’ve also noticed that the general . . .,” Twitter, May 21, 2019, 7:06 a.m., <https://twitter.com/sindresorhus/status/1130792040420167681>.

59 Sindre Sorhus (@sindresorhus), “Don’t let this be a deterrent . . .,” Twitter, May 21, 2019, 9:25 a.m., <https://twitter.com/sindresorhus/status/1130826866921594880>.

60 Sindre Sorhus (@sindresorhus), “Just keep in mind that my time . . .,” Twitter, May 21, 2019, 9:30 a.m., <https://twitter.com/sindresorhus/status/1130828104178339840>.

61 Nolan Lawson, “What It Feels Like to Be an Open-Source Maintainer,” *Read the Tea Leaves*, March 5, 2017, <https://nolanlawson.com/2017/03/05/what-it-feels-like-to-be-an-open-source-maintainer/>.

62 Chris Mayer, “Has the Apache Open Source Vision Become Blurred?,” *JAXenter*, November 25, 2011, <https://jaxenter.com/has-the-apache-open-source-vision-become-blurred-103947.html>.

63 Bryan Clark, “Apache Software Foundation Joins GitHub Open Source Community,” *The GitHub Blog*, GitHub, April 29, 2019, <https://github.blog/2019-04-29-apache-joins-github-community/>

02

64 Julia Evans, “Figuring Out How to Contribute to Open Source,” *Julia Evans* (blog), n.d., <https://jvns.ca/blog/2017/08/06/contributing-to-open-source/>.

65 “Ptychobranchus Subtentum,” *Wikipedia*, last updated April 7, 2019, https://en.wikipedia.org/wiki/Ptychobranchus_subtentum.

[**66**](#) “Reykjavík,” *Wikipedia*, last updated March 27, 2020,
<https://en.wikipedia.org/wiki/Reykjav%C3%ADk>.

[**67**](#) “China: Provinces and Major Cities,” City Population, December 21, 2019, <https://www.citypopulation.de/en/china/cities/>.

[**68**](#) “PEP 0 -- Index of Python Enhancement Proposals (PEPs),” Python.org, accessed March 31, 2020,
<https://www.python.org/dev/peps/>.

[**69**](#) “Proposing Changes to Go,” Golang / Proposal Code, GitHub, accessed March 31, 2020, <https://github.com/golang/proposal/>

[**70**](#) “How Brett Cannon Uses GitHub,” Customer Stories, GitHub, n.d., <https://github.com/customer-stories/brettcannon>.

[**71**](#) “DebianMaintainer,” *Debian Wiki*, last updated August 24, 2019, <https://wiki.debian.org/DebianMaintainer>.

[**72**](#) Stuart Sierra, “Clojure Governance and How It Got That Way,” Clojure, February 17, 2012,
<https://clojure.org/news/2012/02/17/clojure-governance>.

[**73**](#) “Mutate Regexp Body,” Mutant Pull Requests, GitHub, April 18, 2016, <https://github.com/mbj/mutant/pull/565#issuecomment-211498398>.

[**74**](#) Lorenzo Sciandra, “Chain React 2019 - Lorenzo Sciandra - All Hands on Deck - The React Native Community Experience,” Infinite Red, July 31, 2019, YouTube video, <https://www.youtube.com/watch?v=OVzMw3vYrDI&feature=youtu.be>.

75 “README,” Pgcli Code, GitHub, accessed March 31, 2020,
<https://github.com/dbcli/pgcli>.

76 “README,” WP-CLI Code, GitHub, accessed March 31, 2020,
<https://github.com/wp-cli/wp-cli>.

77 “README,” Is-sorted Code, GitHub, n.d.,
<https://github.com/dcousens/is-sorted>.

78 “This Is Python Version 3.9.0 Alpha 5,” Cpython Code, GitHub, accessed March 31, 2020, <https://github.com/python/cpython>.

79 Python / Cpython, GitHub, screenshot taken February 23, 2020,
<https://github.com/python/cpython>.

80 “Getting Started,” React, n.d., <https://reactjs.org/docs/getting-started.html>.

81 Ryan Dahl, “Ryan Dahl - History of Node.js,” Phx Tag Soup, October 5, 2011, YouTube video, 24:48,
<https://www.youtube.com/watch?v=SAc0vQCC6UQ>.

82 Nathan Marz, “History of Apache Storm and Lessons Learned,” *Thoughts from the Red Planet*, October 6, 2014,
<http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html>.

83 Arfon Smith, “The Shape of Open Source,” *The GitHub Blog*, GitHub, June 23, 2016, <https://github.blog/2016-06-23-the-shape-of-open-source/>.

84 Felix Krause, “Scaling Open Source Communities,” *Felix Krause* (blog), January 31, 2017, <https://krausefx.com/blog/scaling-open-source-communities>.

85 “Core Team,” Webpack Code, GitHub, accessed March 31, 2020, <https://github.com/webpack/webpack#core-team>.

86 Youtube-dl Code, GitHub, accessed March 31, 2020, <https://github.com/ytdl-org/youtube-dl>.

87 Font Awesome Code, GitHub, accessed March 31, 2020, <https://github.com/FortAwesome/Font-Awesome>.

88 McKenzie, Sebastian (@sebmck), “GitHub is such a poor tool . . . ,” Twitter, November 16, 2015, 3:47 a.m., <https://twitter.com/sebmck/status/667097915605708804>.

89 “September 19,” Babel / Notes Code, GitHub, September 19, 2016, <https://github.com/babel/notes/blob/master/2016/2016-09/september-19.md>.

90 Stack Overflow Insights, “Developer Survey Results 2019,” Stack Overflow, 2019, <https://insights.stackoverflow.com/survey/2019#technology>.

91 Nadia Eghbal, “Understanding User Support Systems in Open Source,” Nadia Eghbal, September 27, 2018, <https://nadiaeghbal.com/user-support>.

92 Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa, “More Common Than You Think: An In-Depth Study of Casual

Contributors,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Suita, Japan: IEEE, March 2016): 518–528, <https://doi.org/10.1109/saner.2016.68>.

93 Mikeal Rogers, “Building a Better Node Community,” *Node & JavaScript*, Medium, October 10, 2014, <https://medium.com/node-js-javascript/building-a-better-node-community-3f8f45b45cb5#.b2ebksumt>.

94 Pieter Hintjens, “Why Optimistic Merging Works Better,” *Hintjens* (blog), November 16, 2015, <http://hintjens.com/blog:106>.

95 Astropy Code, GitHub, accessed March 31, 2020, <https://github.com/astropy/astropy>.

96 Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi, “Magnet or Sticky? An OSS Project-by-Project Typology,” in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, chair Premkumar Devanbu (Hyderabad, India: Association for Computing Machinery, May 2014): 344–47, <https://doi.org/10.1145/2597073.2597116>.

97 Nicole Carpenter, “The Gentle Side of Twitch,” *Gizmodo*, April 23, 2019, <https://gizmodo.com/the-gentle-side-of-twitch-1834215442>.

98 Ssh-chat Code, GitHub, accessed March 31, 2020, <https://github.com/shazow/ssh-chat>.

99 Spencer Heath MacCallum, *The Art of Community* (Menlo Park, CA: Institute for Humane Studies, 1970), 5.

100 MacCallum, *The Art of Community*, 66.

101 T. L. Taylor, *Watch Me Play: Twitch and the Rise of Game Live Streaming* (Princeton, NJ: Princeton University Press, 2018), 92–93.

102 MacCallum, *The Art of Community*, 67.

103 Nadia Eghbal, “Emerging Models for Open Source Contributions” (presentation, GitHub CodeConf, Los Angeles, June 29, 2016),
<https://www.slideshare.net/NadiaEghbal/emerging-models-for-open-source-contributions>.

104 Mikeal Rogers, “Healthy Open Source,” *Node.js Collection*, Medium, February 22, 2016, <https://medium.com/the-node-js-collection/healthy-open-source-967fa8be7951>.

105 Taylor Wofford, “Fuck You and Die: An Oral History of Something Awful,” *Vice*, April 5, 2017,
https://www.vice.com/amp/en_us/article/nzg4yw/fuck-you-and-die-an-oral-history-of-something-awful.

106 Adam Rowe, “Why Paid Apps Could Be the Future of Online Communities,” *Tech.co*, November 1, 2019,
<https://tech.co/news/woolfer-paid-app-online-communities-2019-11>.

107 Kevin Simler, “Border Stories,” *Melting Asphalt*, March 2, 2015,
<https://meltingasphalt.com/border-stories/>.

03

108 Star Simpson (@starsandrobots), “Til recently you were online . . .,” Twitter, November 5, 2017, 6:54 p.m.,
<https://twitter.com/starsandrobots/status/927323260244463616>.

109 Ronald Coase, “The Nature of the Firm,” *Economica* 4, no. 16 (November 1937): 386–405,
<https://doi.org/10.1017/cbo9780511817410.009>.

110 Elinor Ostrom, *Governing the Commons: The Evolution of Institutions for Collective Action* (Cambridge: Cambridge University Press, 1990), Loc 2053.

111 Yochai Benkler, “Coase’s Penguin, Or, Linux and ‘The Nature of the Firm,’” *The Yale Law Journal* 112, no. 3 (2002): 369–446,
<https://doi.org/10.2307/1562247>.

112 Benkler, “Coase’s Penguin,” 381.

113 Guido van Rossum, “Foreword for ‘Programming Python’ (1st Ed.),” Python.org, May 1996,
<https://www.python.org/doc/essays/foreword/>.

114 Linus Torvalds, “LINUX’s History,” Carnegie Mellon University School of Computer Science, July 31, 1992,
<https://www.cs.cmu.edu/~awb/linux.history.html>.

115 Linus Torvalds, “Re: Kernel SCM Saga..,” Mailing List ARChive, April 7, 2005, <https://marc.info/?l=linux->

[kernel&m=111288700902396.](#)

[116](#) Benkler, “Coase’s Penguin,” 378.

[117](#) M. D. McIlroy, E. N. Pinson, and B. A. Tague, “UNIX Time-Sharing System: Foreword,” *The Bell System Technical Journal* 57, no. 6 (1978): 1902, <https://doi.org/10.1002/j.1538-7305.1978.tb02135.x>.

[118](#) Benkler, “Coase’s Penguin,” 379.

[119](#) David Heinemeier Hansson, “The Perils of Mixing Open Source and Money,” November 12, 2013, <https://dhh.dk/2013/the-perils-of-mixing-open-source-and-money.html>.

[120](#) Josh Lerner and Jean Tirole, “The Simple Economics of Open Source,” NBER Working Paper 7600, National Bureau of Economic Research, March 2000, 32, <https://doi.org/10.3386/w7600>.

[121](#) Eugene Wei, “Status as a Service (StaaS),” *Remains of the Day*, February 19, 2019, <https://www.eugenewei.com/blog/2019/2/19/status-as-a-service>.

[122](#) Michael Wesch, “YouTube and You: Experiences of Self-Awareness in the Context Collapse of the Recording Webcam,” *Explorations in Media Ecology* 8, no. 2 (2009): 19–34.

[123](#) Robert E. Kraut and Paul Resnick, *Building Successful Online Communities: Evidence-Based Social Design* (Cambridge, MA: The MIT Press, 2016), 165.

[124](#) Kraut and Resnick, *Building Successful Online Communities*, 128.

[125](#) Coraline Ada Ehmke (CoralineAda), “Transphobic Maintainer Should Be Removed from Project,” Opal Issues, GitHub, June 18, 2015, <https://github.com/opal/opal/issues/941>.

[126](#) Strand McCutchen (strand), “Create a Code of Conduct,” Opal Issues, GitHub, June 21, 2015, <https://github.com/opal/opal/issues/942>.

[127](#) Kraut and Resnick, *Building Successful Online Communities*, 128.

[128](#) Kraut and Resnick, *Building Successful Online Communities*, 217.

[129](#) Kraut and Resnick, *Building Successful Online Communities*, 165.

[130](#) Jamie Kyle (jamiebuilds), “(REVERTED): Add Text to MIT License Banning ICE Collaborators,” Lerna Pull Requests, GitHub, August 29, 2018, <https://github.com/lerna/lerna/pull/1616>.

[131](#) Daniel Stockman (evocateur), “Chore: Restore Unmodified MIT License,” Lerna Pull Request, GitHub, August 30, 2018, <https://github.com/lerna/lerna/pull/1633>.

[132](#) Pete Resnick, “On Consensus and Humming in the IETF,” IETF Tools, June 2014, <https://tools.ietf.org/html/rfc7282>.

[133](#) “Viewing a Project’s Contributors,” GitHub Help, n.d., <https://help.github.com/en/articles/viewing-a-projects-contributors>.

[134](#) “Viewing Contributions on Your Profile,” GitHub Help, n.d., <https://help.github.com/en/github/setting-up-and-managing-your-github-profile/viewing-contributions-on-your-profile>.

135 Dan Abramov (gaearon) Overview, Github screenshot, 2020,
<https://github.com/gaearon>.



136 “Emoji Key (and Contribution Types),” All Contributors, n.d., <https://allcontributors.org/docs/en/emoji-key>.

137 All-Contributors Code, GitHub, accessed May 1, 2020, <https://github.com/all-contributors/all-contributors>.

138 Charlotte Hess and Elinor Ostrom, “A Framework for Analyzing the Knowledge Commons,” in *Understanding Knowledge as a Commons: From Theory to Practice*, eds. Hess and Ostrom (Cambridge, MA: MIT Press, 2011), 48.

139 Benjamin Lupton (balupton), “Help Open-Source Maintainers Stay Sane,” Isaacs / Github Issues, GitHub, April 12, 2014, <https://github.com/isaacs/github/issues/167>.

140 Frederick Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed. (Reading: Addison-Wesley, 1995), 32.

141 “Teams,” Django Software Foundation, accessed March 31, 2020, <https://www.djangoproject.com/foundation/teams/>.

142 Caddyserver / Caddy, GitHub, accessed March 31, 2020, <https://github.com/caddyserver/caddy>.

143 Spencer Heath MacCallum, *The Art of Community* (Menlo Park, CA: Institute for Humane Studies, 1970), 63–67.

144 MacCallum, *The Art of Community*, 63.

145 “Meet the Team,” Babel, accessed March 31, 2020,
<https://babeljs.io/team>.

146 Jacob Kaplan-Moss, “Retiring as BDFLs,” *Jacob Kaplan-Moss* (blog), January 13, 2014, <https://jacobian.org/2014/jan/13/retiring-as-bdfls/>.

147 Urllib3, GitHub, accessed March 13, 2020,
<https://github.com/urllib3/urllib3/>.

148 Andrey Petrov, “How to Hand over an Open Source Project to a New Maintainer,” Medium, February 9, 2018,
<https://medium.com/@shazow/how-to-hand-over-an-open-source-project-to-a-new-maintainer-db433aaf57e8>.

149 Clint Finley, “Giving Open-Source Projects Life after a Developer’s Death,” *Wired*, November 6, 2017,
<https://www.wired.com/story/giving-open-source-projects-life-after-a-developers-death/>.

150 Alanna Irving, “Funding Open Source: How Webpack Reached \$400k+/Year,” *Open Collective*, October 23, 2017,
<https://medium.com/open-collective/funding-open-source-how-webpack-reached-400k-year-dfb6d8384e19>.

151 Christopher Hiller, Nadia Eghbal, and Mikeal Rogers, “Maintaining a Popular Project and Managing Burnout with Christopher Hiller,” *Request for Commits*, podcast audio, November 1, 2017, <https://changelog.com/rfc/15>.

[152](#) Ayrton Sparling (FallingSnow), “I Dont Know What to Say,” Event-stream Issues, GitHub, November 20, 2018,
<https://github.com/dominictarr/event-stream/issues/116>.

[153](#) Dominic Tarr (dominictarr), “Statement on Event-Stream Compromise,” Dominictarr / Readme.md Code, GitHub, November 26, 2018,
<https://gist.github.com/dominictarr/9fd9c1024c94592bc7268d36b8d83b3a>.

[154](#) Felix Geisendörfer, “The Pull Request Hack,” *Felix Geisendörfer* (blog), March 11, 2013, <https://felixge.de/2013/03/11/the-pull-request-hack.html>.

[155](#) Na Sun, Patrick Pei-Luen Rau, and Liang Ma, “Understanding Lurkers in Online Communities: A Literature Review,” *Computers in Human Behavior*, no. 38 (September 2014): 110–117,
<https://www.sciencedirect.com/science/article/pii/S0747563214003008>

.

[156](#) Kraut and Resnick, *Building Successful Online Communities*, 63.

[157](#) Andrew J. Ko and Parmit K. Chilana, “How Power Users Help and Hinder Open Bug Reporting,” in *Proceedings of the 28th International Conference on Human Factors in Computing Systems - CHI 10*, chair Elizabeth Mynatt (New York: Association for Computing Machinery, April 2010): 1665–74,
<https://doi.org/10.1145/1753326.1753576>.

158 Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies, “Why Software Projects Need Heroes (Lessons Learned from 1000+ Projects),” ArXiv, April 22, 2019, <https://arxiv.org/pdf/1904.09954.pdf>.

159 Minghui Zhou and Audris Mockus, “What Make Long Term Contributors: Willingness and Opportunity in OSS Community,” in *2012 34th International Conference on Software Engineering (ICSE)* (Zurich: IEEE, June 2012): 518–28, <https://doi.org/10.1109/icse.2012.6227164>.

160 Zhou and Mockus, “What Make Long Term Contributors,” 518.

161 Naomi Ceder, “Come for the Language, Stay for the Community” (presentation, EuroPython 2016, Bilbao, Spain, July 21, 2016), <https://ep2016.europython.eu/media/conference/slides/keynote-stay-for-the-community.pdf>.

162 Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi, “Magnet or Sticky? An OSS Project-by-Project Typology,” in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, chair Premkumar Devanbu (Hyderabad, India: Association for Computing Machinery, May 2014): 344–46, <https://doi.org/10.1145/2597073.2597116>.

163 Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa, “More Common Than You Think: An In-Depth Study of Casual Contributors,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Suita,

Japan: IEEE, March 2016), 518–528,
<https://doi.org/10.1109/saner.2016.68>.

164 Igor Steinmacher, Igor Wiese, Ana Paula Chaves, and Marco Aurélio Gerosa, “Why Do Newcomers Abandon Open Source Software Projects?,” in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)* (IEEE: San Francisco, May 2013): 31,
<https://doi.org/10.1109/chase.2013.6614728>.

165 Josh Lerner and Jean Tirole, “Some Simple Economics of Open Source,” *The Journal of Industrial Economics* 50, no. 2 (June 2002): 197–234, <https://doi.org/10.1111/1467-6451.00174>.

166 Nadia Eghbal, “The Rise of Few-Maintainer Projects,” *Increment* 9 (May 2019), <https://increment.com/open-source/the-rise-of-few-maintainer-projects/>.

167 Kraut and Resnick, *Building Successful Online Communities*, 205.

168 Christoph Hannebauer, Matthias Book, and Volker Gruhn, “An Exploratory Study of Contribution Barriers Experienced by Newcomers to Open Source Software Projects,” in *Proceedings of the 1st International Workshop on CrowdSourcing in Software Engineering - CSI-SE 2014*, chairs Gordon Fraser et al. (Hyderabad: Association for Computing Machinery, June 2014): 11–14,
<https://doi.org/10.1145/2593728.2593732>.

169 Ko and Chilana, “How Power Users Help and Hinder Open Bug Reporting.”

170 Owen Williams, “The Internet Relies on People Working for Free,” *OneZero*, Medium, September 16, 2019,
<https://onezero.medium.com/the-internet-relies-on-people-working-for-free-a79104a68bcc>.

171 Jeff Forcier (@bitprophet), “TIL that SpaceX . . .,” Twitter, February 7, 2018, 11:34 p.m.,
<https://twitter.com/bitprophet/status/961095769599234049>.

172 “Anonymous Aggregate User Behaviour Analytics,” Homebrew Documentation, n.d., <https://docs.brew.sh/Analytics>.

173 Sneak, “Why are you even . . .,” Hacker News, comment, April 26, 2016, <https://news.ycombinator.com/item?id=11570483>.

174 Feross Aboukhadijeh (feross), “Which Companies Are Using ‘Standard’?”, Standard Issues, GitHub, May 9, 2018,
<https://github.com/standard/standard/issues/744>.

175 Ko and Chilana, “How Power Users Help and Hinder Open Bug Reporting,” 1665.

176 K. Crowston and J. Howison, “Assessing the Health of Open Source Communities,” *Computer* 39, no. 5 (May 2006): 89–91,
<https://doi.org/10.1109/mc.2006.152>.

177 Nadia Eghbal, “What Success Really Looks Like in Open Source,” Medium, February 5, 2016,
<https://medium.com/@nayafia/what-success-really-looks-like-in-open-source-2dd1facaf91c>.

178 Nadia Eghbal, “Methodologies for Measuring Project Health,” Nadia Eghbal, July 18, 2018, <https://nadiaeghbal.com/project-health>.

179 Brooks, *The Mythical Man-Month*, 16.

180 Richard Schneeman, “Saving Sprockets,” *Schneems*, May 31, 2016, <https://www.schneems.com/2016/05/31/saving-sprockets.html>.

181 “Open Source Metrics,” Open Source Guides, n.d., <https://opensource.guide/metrics/>.

182 Eghbal, “Methodologies for Measuring Project Health.”

04

183 Kevin Kelly, “Immortal Technologies,” *The Technium*, February 9, 2006, <https://kk.org/thetechnium/immortal-techno/>.

184 Nathan Ensmenger, “When Good Software Goes Bad: The Surprising Durability of an Ephemeral Technology,” Indiana University, September 11, 2014, <http://homes.sice.indiana.edu/nensmeng/files/ensmenger-mice.pdf>.

185 Fergus Henderson, “Software Engineering at Google,” ArXiv, February 19, 2019, <https://arxiv.org/pdf/1702.01715.pdf>.

186 Alex Handy, “Ruby on Rails 3.0 Goes Modular,” *SD Times*, February 12, 2010, <https://sdtimes.com/ruby-on-rails/ruby-on-rails-3-0-goes-modular>.

[187](#) Yehuda Katz, “Rails and Merb Merge,” *Katz Got Your Tongue*, December 23, 2008, <https://yehudakatz.com/2008/12/23/rails-and-merb-merge/>.

[188](#) Byrne Hobart, “The Case for Subsidizing, or Banning, COBOL Classes,” Medium, March 29, 2019,
<https://medium.com/@byrnehobart/you-cant-reduce-all-economic-decisions-to-a-series-of-financial-bets-but-it-s-a-good-way-to-d40e88e89e17>.

[189](#) Chris Zacharias, “A Conspiracy To Kill IE6,” *Chris Zacharias* (blog), May 1, 2019, <http://blog.chriszacharias.com/a-conspiracy-to-kill-ie6>.

[190](#) Jacob Friedmann, “SmooshGate: The Ongoing Struggle between Progress and Stability in JavaScript,” Medium, March 10, 2018,
<https://medium.com/@jacobdfriedmann/smooshgate-the-ongoing-struggle-between-progress-and-stability-in-javascript-2a971c1162dd>.

[191](#) Michael Ficarra (michaelficarra), “Rename Flatten to Smoosh,” Tc39 / Proposal-flatMap Pull Requests, GitHub, March 6, 2018,
<https://github.com/tc39/proposal-flatMap/pull/56>.

[192](#) Neal Stephenson, *In the Beginning . . . Was the Command Line* (New York: William Morrow Paperbacks, 1999), 88.

[193](#) Chrislgarry / Apollo-11, GitHub, accessed March 31, 2020,
<https://github.com/chrislgarry/Apollo-11>.

[194](#) Nadia Eghbal, “The Hidden Costs of Software” (lecture, the Web Conference, San Francisco, May 16, 2019).

195 Free Software Foundation, “What Is Free Software?,” GNU Operating System, July 30, 2019, <https://www.gnu.org/philosophy/free-sw.en.html>.

196 Jacob Thornton, “What Is Open Source & Why Do I Feel So Guilty?,” Dotconferences, November 30, 2012, Youtube video, 20:15, <https://www.youtube.com/watch?v=UIDb6VBO9os>.

197 Roy Revelt (revelt), “But guys, this package is not . . .,” Core-js Issues comment, GitHub, June 13, 2019, <https://github.com/zloirock/core-js/issues/571#issuecomment-501661663>.

198 Tristanleboss, “@revelt Forking is one thing . . .,” Core-js Issues comment, GitHub, June 13, 2019, <https://github.com/zloirock/core-js/issues/571#issuecomment-501889710>.

199 Denis Pushkarev (zloirock), “@revelt please, don’t say me what I should do . . .,” Core-js Issues comment, GitHub, June 14, 2019, <https://github.com/zloirock/core-js/issues/571#issuecomment-502040557>.

200 Norbert Wiener, *The Human Use of Human Beings* (Boston: Houghton Mifflin Company, 1950), 129.

201 Spencer Heath MacCallum, *The Art of Community* (Menlo Park, CA: Institute for Humane Studies, 1970), 48.

202 David Heinemeier Hansson, “Open Source beyond the Market,” *Signal v. Noise*, May 20, 2019, <https://m.signalvnoise.com/open-source-beyond-the-market>.

203 David Bollier, “The Growth of the Commons Paradigm,” in *Understanding Knowledge as a Commons: From Theory to Practice*, eds. Charlotte Hess and Elinor Ostrom (Cambridge, MA: MIT Press, 2011), 34.

204 Donald Stufft (@dstufft), “PyPI ‘costs’ like 2-3 million dollars. . .,” Twitter, May 11, 2019, 5:10 p.m.,
<https://twitter.com/dstufft/status/1127320131359653890>.

205 Donald Stufft (@dstufft), “The first full month of PyPI/PSF . . .,” Twitter, July 21, 2017, 1:29 p.m.,
<https://twitter.com/dstufft/status/888450899357704192>.

206 Donald Stufft (@dstufft), “April ‘bill’ for Fastly . . .,” Twitter, May 11, 2019, 5:26 p.m.,
<https://twitter.com/dstufft/status/1127324217622638599>.

207 Drew DeVault, “The Path to Sustainably Working on FOSS Full-Time,” *Drew DeVault’s Blog*, February 24, 2018,
<https://drewdevault.com/2018/02/24/The-road-to-sustainable-FOSS.html>.

208 Werner Vogels, “Eventually Consistent,” *Communications of the ACM* 52, no. 1 (January 2009): 40,
<https://doi.org/10.1145/1435417.1435432>.

209 Lily Hay Newman, “GitHub Survived the Biggest DDoS Attack Ever Recorded,” *Wired*, March 1, 2018,
<https://www.wired.com/story/github-ddos-memcached/>

210 Meira Gebel, “In 15 Years Facebook Has Amassed 2.3 Billion Users - More Than Followers of Christianity,” *Business Insider*, February 4, 2019, <https://www.businessinsider.com/facebook-has-2-billion-plus-users-after-15-years-2019-2>.

211 Barry Schwartz, “Google: We Can’t Have Customer Service Because . . .,” Search Engine Roundtable, August 24, 2011, <https://www.seroundtable.com/google-support-staff-limits-13916.html>.

212 Nolan Lawson, “What It Feels Like to Be an Open-Source Maintainer,” *Read the Tea Leaves*, March 5, 2017, <https://nolanlawson.com/2017/03/05/what-it-feels-like-to-be-an-open-source-maintainer/>.

213 Frederick Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed. (Reading: Addison-Wesley, 1995), 121.

214 Devon Zuegel, “The City Guide to Open Source,” *Increment* 9, May 2019, <https://increment.com/open-source/the-city-guide-to-open-source/>.

215 Robert Glass, *Facts and Fallacies of Software Engineering* (Boston: Addison-Wesley, 2010), 174.

216 “Section 230 of the Communications Decency Act,” Electronic Frontier Foundation, n.d., <https://www.eff.org/issues/cda230>.

217 Ben Balter, “Open Source License Usage on GitHub.com,” *The GitHub Blog*, GitHub, March 9, 2015, <https://github.blog/2015-03-09-open-source-license-usage-on-github-com/>.

[218 “The MIT License,” Open Source Initiative, n.d.,
<https://opensource.org/licenses/MIT>.](https://opensource.org/licenses/MIT)

[219 “Open Source Survey,” Open Source Survey, 2017,
<https://opensourcesurvey.org/2017/>.](https://opensourcesurvey.org/2017/)

[220 “The Developer Coefficient: A \\$300B Opportunity for Business,” Stripe, September 2018, <https://stripe.com/reports/developer-coefficient-2018>.](https://stripe.com/reports/developer-coefficient-2018)

[221 Ensmenger, “When Good Software Goes Bad.”](#)

[222 “About Required Status Checks,” GitHub Help, n.d.,
<https://help.github.com/en/github/administering-a-repository/about-required-status-checks>.](https://help.github.com/en/github/administering-a-repository/about-required-status-checks)

[223 Alan Zeino, “Faster Together: Uber Engineering’s IOS Monorepo,” *Uber Engineering* \(blog\), March 6, 2017,
<https://eng.uber.com/ios-monorepo/>](https://eng.uber.com/ios-monorepo/)

[224 Steve Klabnik \(@steveklabnik\), “Today I glanced at some numbers . . .,” Twitter, June 14, 2019, 11:17 a.m.,
<https://twitter.com/steveklabnik/status/1139552342842458112>.](https://twitter.com/steveklabnik/status/1139552342842458112)

[225 “The State of the Octoverse,” GitHub, 2019,
<https://octoverse.github.com/>.](https://octoverse.github.com/)

[226 Russ Cox, “Our Software Dependency Problem,” *Research!rsc*, January 23, 2019, <https://research.swtch.com/deps>.](https://research.swtch.com/deps)

227 Dan Goodin, “Failure to Patch Two-Month-Old Bug Led to Massive Equifax Breach,” *Ars Technica*, September 13, 2017, <https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/>.

228 Clayton Christensen, *The Innovator’s Dilemma: The Revolutionary Book That Will Change the Way You Do Business* (New York, NY: Harper Business, 2003).

229 “PyPy Features,” PyPy, n.d., <https://pypy.org/features.html>.

230 “Call for Donations - PyPy to Support Python3!,” PyPy, December 2019, <https://web.archive.org/web/20191209173423/https://www.pypy.org/pypy3donate.html>.

231 Mikeal Rogers, “Request’s Past, Present and Future,” Request Issues, GitHub, March 30, 2019, <https://github.com/request/request/issues/3142>.

232 “Moving to Require Python 3,” Python 3 Statement, accessed March 30, 2020, <https://python3statement.org/>.

233 VPeric, “Differences between Distribute, Distutils, Setuptools and Distutils2?,” Stack Overflow Questions, accessed March 2, 2020, <https://stackoverflow.com/questions/6344076/differences-between-distribute-distutils-setuptools-and-distutils2>.

234 Nick Heath, “Python Is Eating the World: How One Developer’s Side Project Became the Hottest Programming Language on the Planet,” *TechRepublic*, August 6, 2019,

<https://www.techrepublic.com/article/python-is-eating-the-world-how-one-developers-side-project-became-the-hottest-programming-language-on-the-planet/>.

235 Clint Finley, “GitHub ‘Sponsors’ Now Lets Users Back Open Source Projects,” *Wired*, May 23, 2019,

<https://www.wired.com/story/github-sponsors-lets-users-back-open-source-projects/>.

236 J. Bradford De Long and A. Michael Froomkin, “The Next Economy?” (draft), University of Miami School of Law, April 6, 1997, <http://osaka.law.miami.edu/~froomkin/articles/newecon.htm>.

237 Ben Thompson, “AWS, MongoDB, and the Economic Realities of Open Source,” *Stratechery*, January 14, 2019,

<https://stratechery.com/2019/aws-mongodb-and-the-economic-realities-of-open-source/>.

238 Bill Gates, “An Open Letter to Hobbyists,” February 3, 1976, via Wikimedia Commons,

https://commons.wikimedia.org/wiki/File:Bill_Gates_Letter_to_Hobbyists.jpg.

239 David Friedman, *Price Theory: an Intermediate Text* (Cincinnati, OH: South-Western Publishing Co, 1986), 20.

240 Ben Lesh (@BenLesh), “Open Source is such a strange thing . . . ,” Twitter, November 30, 2017, 1:26 p.m.,

<https://twitter.com/BenLesh/status/936300388906446848>.

241 Jane Jacobs, *The Death and Life of Great American Cities* (New York: Vintage Books, 1992), 433.

242 Timothy Patitsas, Nadia Eghbal, and Henry Zhu, “City as Liturgy,” *Hope in Source*, podcast audio, March 21, 2019, <https://hopeinsource.com/city/>.

243 Randall W. Eberts, “White Paper on Valuing Transportation Infrastructure,” W.E. Upjohn Institute for Employment Research, January 1, 2014, 10, <https://research.upjohn.org/cgi/viewcontent.cgi?httpsredir=1&article=1217&context=reports>.

244 Azer Koçulu, “I’ve Just Liberated My Modules,” *Azer Koçulu* (blog), March 23, 2016, <https://kodfabrik.com/journal/i-ve-just-liberated-my-modules/>.

245 “@Babel/core,” Npm, accessed December 2019, <https://www.npmjs.com/package/@babel/core>.

246 Isaac Schlueter (@Izs), “Kik, Left-Pad, and Npm,” *The Npm Blog*, March 23, 2016, <https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm>.

247 HM Treasury, “Valuing Infrastructure Spend: Supplementary Guidance to the Green Book,” March 2015, 3, https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/417822/PU1798_Valuing_Infrastructure_Spend_-_lastest_draft.pdf.

248 “Sindre Sorhus Is Creating Open Source Software,” accessed March 13, 2020, Patreon, <https://www.patreon.com/sindresorhus>.

249 Benjamin E. Coe (bcoe), “Npm Users by Downloads,” Npm-top.md Code, GitHub, June 6, 2018,
<https://gist.github.com/bcoe/dcc961b869bbf6685002>.

250 “Sindre Sorhus: Patreon Earnings Statistics Graphs Rank,” Graphtreon, accessed March 31, 2020,
<https://graphtreon.com/creator/sindresorhus>.

251 Sindre Sorhus, “Answering Anything & Everything,” *Sindre Sorhus’ Blog*, Medium, August 10, 2016,
<https://blog.sindresorhus.com/answering-anything-678ce5623798>.

252 Vivian Cromwell, “Between the Wires: An Interview with Open Source Developer Sindre Sorhus,” FreeCodeCamp, September 4, 2017, <https://www.freecodecamp.org/news/sindre-sorhus-8426c0ed785d/>.

253 Donald Stufft, “Hire Me,” Donald Stufft, October 17, 2016,
<https://caremad.io/posts/2016/10/hire-me/>.

254 Donald Stufft, “A New Home,” Donald Stufft, January 18, 2017,
<https://caremad.io/posts/2017/01/a-new-home/>.

255 “Drew DeVault Is Creating Free Software,” Patreon, accessed March 31, 2020, <https://www.patreon.com/sircmpwn>.

256 “Evan Is Creating Vue.js,” Patreon, accessed March 13, 2020,
<https://patreon.com/evanyou>.

257 Evan You, “Vue Is Now on OpenCollective!,” *The Vue Point*, Medium, September 11, 2017, <https://medium.com/the-vue-point/vue->

[is-now-on-opencollective-1ef89ca1334b.](#)

258 Sophie Alpert (@sophiebits), “Is this what it feels like . . .,” Twitter, November 10, 2019, 7:27 p.m.,
[https://twitter.com/sophiebits/status/1193686560413106177.](https://twitter.com/sophiebits/status/1193686560413106177)

259 Owen Phillips, “The Anonymous MVP of the NBA Finals,” *The Outline*, June 12, 2017, <https://theoutline.com/post/1706/the-anonymous-mvp-of-the-nba-finals-velocityraps-illegal-streaming>.

260 Ryan Regier, “We Are in a Golden Age of Illegal Sports Streaming and It’s Showing Us How Copyright Infringement Can Result in Better Content,” Medium, January 20, 2019,
[https://medium.com/@ryregier/we-are-in-a-golden-age-of-illegal-sports-streaming-and-its-showing-us-how-copyright-infringement-d835ae291ed2.](https://medium.com/@ryregier/we-are-in-a-golden-age-of-illegal-sports-streaming-and-its-showing-us-how-copyright-infringement-d835ae291ed2)

05

261 Jane Jacobs, *The Death and Life of Great American Cities* (New York: Vintage Books, 1992), 55.

262 Office of the Under Secretary of Defense (Comptroller), “National Defense Budget Estimates for FY2020,” United States Department of Defense, May 2019,

https://comptroller.defense.gov/Portals/45/Documents/defbudget/fy2020/FY20_Green_Book.pdf.

263 Guido van Rossum, “[Python-Committers] Transfer of Power,” The Python-Committers Archives, July 12, 2018,
<https://mail.python.org/pipermail/python-committers/2018-July/005664.html>.

264 Jake Edge, “PEP 572 and Decision-Making in Python,” *LWN.net*, June 20, 2018, <https://lwn.net/Articles/757713/>.

265 Guido van Rossum, “A Different Way to Focus Discussions,” *LWN.net*, May 18, 2018, <https://lwn.net/Articles/759557/>.

266 Jonathan Zdziarski, “On the State of Open Source,” Zdziarski’s *Blog of Things*, October 3, 2016, <https://www.zdziarski.com/blog/?p=6296>.

267 Kristen Roupenian, “What It Felt Like When ‘Cat Person’ Went Viral,” *The New Yorker*, January 9, 2019,
<https://www.newyorker.com/books/page-turner/what-it-felt-like-when-cat-person-went-viral>.

268 “About,” Lobsters, n.d., <https://lobste.rs/about>.

269 “Product Hunt Pro Tips,” Product Hunt, n.d.,
<https://www.producthunt.com/protips>.

270 Zdziarski, “On the State of Open Source.”

271 C. Titus Brown, “How Open Is Too Open?,” *Living in an Ivory Basement*, June 26, 2018, <http://ivory.idyll.org/blog/2018-how-open-is-too-open.html>.

272 Withoutboats, “Organizational Debt,” *Withoutblogs*, December 16, 2018, <https://boats.gitlab.io/blog/post/rust-2019>.

273 Brown, “How Open Is Too Open?”

274 Rich Hickey (richhickey), “I found out about this diatribe . . . ,” R/Clojure comment, Reddit, October 7, 2017,
https://www.reddit.com/r/Clojure/comments/73yznc/on_whose_authority/.

275 Donald Hicks and David Gasca, “A Healthier Twitter: Progress and More to Do,” *Twitter* (blog), April 16, 2019,
https://blog.twitter.com/en_us/topics/company/2019/health-update.html.

276 Anna Wiener, “Jack Dorsey’s TED Interview and the End of an Era,” *The New Yorker*, April 27, 2019,
<https://www.newyorker.com/news/letter-from-silicon-valley/jack-dorseys-ted-interview-and-the-end-of-an-era>.

277 E. Dunham, “Rust’s Community Automation,” *Edunham*, September 27, 2016,
https://edunham.net/2016/09/27/rust_s_community_automation.html.

278 Robert E. Kraut and Paul Resnick, *Building Successful Online Communities: Evidence-Based Social Design* (Cambridge, MA: The MIT Press, 2016), 112.

279 “Dear GitHub,” Dear-Github Code, GitHub, January 10, 2016,
<https://github.com/dear-github/dear-github>.

280 Brandon Keepers (bkeepers), “Dear Open Source Maintainers,” Dear-Github Pull Requests, GitHub, February 12, 2016,
<https://github.com/dear-github/dear-github/pull/115>.

281 Isaacs / Github Code, GitHub, accessed April 25, 2020,
<https://github.com/isaacs/github>.

282 “Rust Highfive Robot,” GitHub, n.d., <https://github.com/rust-highfive>.

283 “Kubernetes Prow Robot,” GitHub, n.d., <https://github.com/k8s-ci-robot>.

284 Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa, “The Power of Bots: Characterizing and Understanding Bots in OSS Projects,” *Proceedings of the ACM on Human-Computer Interaction* 2 (November 2018): 1–19, <https://doi.org/10.1145/3274451>.



285 “The Configuration Issue to End All Configuration Issues,” Probot Issues, GitHub screenshot, November 20, 2017, <https://github.com/probot/probot/issues/258#issuecomment-345739177>.

286 Nadia Eghbal, “Understanding User Support Systems in Open Source,” Nadia Eghbal, September 27, 2018, <https://nadiaeghbal.com/user-support>.

287 “README,” Youtube-dl, GitHub, accessed March 13, 2020, <https://github.com/rg3/youtube-dl>.

288 React-native-firebase Issues, GitHub, accessed March 29, 2020, https://raw.githubusercontent.com/invertase/react-native-firebase/d6db2601f62fa35e79957a6f73454e62e85f9714/.github/ISSUE_TEMPLATE/Bug_report.md.

289 Philip Guo, “PG Vlog #75 - Python Tutor Software Development Philosophy,” October 23, 2017, YouTube video, 10:30, <https://www.youtube.com/watch?v=sVtXLdBRfyE>.

290 Philip Guo, “Ten Years and Nearly Ten Million Users: My Experience Being a Solo Maintainer of Open-Source Software in Academia,” *Philip J. Guo* (blog), November 2019, <http://pgbovine.net/python-tutor-ten-years.htm>.

291 Gabriel Vieira, “Re: A Look at the Design of Lua,” Hacker News, October 30, 2018, <https://news.ycombinator.com/item?id=18327661>.

292 Mike McQuaid, “Stop Mentoring First-Time Contributors,” *Mike McQuaid* (blog), February 16, 2019,

<https://mikemcquaid.com/2019/02/16/stop-mentoring-first-time-contributors/>.

293 Josh Constine, “Instagram Hits 1 Billion Monthly Users, Up from 800M in September,” *TechCrunch*, June 20, 2018,

<https://techcrunch.com/2018/06/20/instagram-1-billion-users/.>

294 Meira Gebel, “In 15 Years Facebook Has Amassed 2.3 Billion Users—More Than Followers of Christianity,” *Business Insider*, February 4, 2019, <https://www.businessinsider.com/facebook-has-2-billion-plus-users-after-15-years-2019-2.>

295 “GitHub Community Guidelines,” GitHub Help, accessed March 13, 2020, <https://help.github.com/en/articles/github-community-guidelines.>

296 Michael Ficarra (michaelficarra), “Rename Flatten to Smoosh,” Tc39 / Proposal-flatMap Pull Requests, GitHub screenshot, May 22, 2018, <https://github.com/tc39/proposal-flatMap/pull/56.>

297 “Guide to Building a Moderation Team,” Twitch, n.d., <https://help.twitch.tv/s/article/guide-to-building-a-moderation-team.>

298 Jessica Rose and John Resig, “Walking Away from Your Open Source Project: John Resig,” *Pursuit Podcast*, YouTube audio only, December 26, 2017, 11:15, <https://youtu.be/K9HGeC2RA-Q.>

299 Eghbal, “Understanding User Support Systems in Open Source.”

300 Sarah T. Roberts, *Behind the Screen: Content Moderation in the Shadows of Social Media* (Yale University Press, 2019), 129.

301 Xin Tan, “Reducing the Workload of the Linux Kernel Maintainers: Multiple-Committer Model,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, chairs Marlon Dumas et al. (Tallinn, Estonia: Association for Computing Machinery, August 2019): 1205–7, <https://doi.org/10.1145/3338906.3342490>.

302 “DEP 0007: Official Django Projects,” Django / Deps Code, GitHub, January 3, 2019, <https://github.com/django/deps/blob/master/final/0007-official-projects.rst>.

303 “Linux Kernel Development Report 2016,” The Linux Foundation, 2016, <https://www2.thelinuxfoundation.org/linux-kernel-development-report-2016>.

304 Henry Zhu, “Planning for 7.0.,” *Babel* (blog), September 12, 2017, <https://babeljs.io/blog/2017/09/12/planning-for-7.0>.

305 Daniel Stenberg, “I’m Leaving Mozilla,” *Daniel Stenberg* (blog), November 18, 2018, <https://daniel.haxx.se/blog/2018/11/18/im-leaving-mozilla/>.

306 Daniel Stenberg, “I’m on Team WolfSSL,” *Daniel Stenberg* (blog), February 2, 2019, <https://daniel.haxx.se/blog/2019/02/02/im-on-team-wolfssl/>.

307 Nadia Eghbal, “Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure,” Ford Foundation, July 14, 2016, <https://www.fordfoundation.org/about/library/reports-and-studies/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure>.

308 Owen Williams, “The Internet Relies on People Working for Free,” *OneZero*, Medium, September 16, 2019, <https://onezero.medium.com/the-internet-relies-on-people-working-for-free-a79104a68bcc>.

309 “Support for Business,” Microsoft Support, February 28, 2019, <https://support.microsoft.com/en-us/help/4341255/support-for-business>.

310 “How the ASF Works,” The Apache Software Foundation, n.d., <https://www.apache.org/foundation/how-it-works.html>.

311 Eric Berry, “Why Funding Open Source Is Hard,” *Hacker Noon*, December 3, 2017, <https://hackernoon.com/why-funding-open-source-is-hard-652b7055569d>.

312 Feross Aboukhadijeh, “Recap of the ‘Funding’ Experiment,” *Feross* (blog), August 28, 2019, <https://feross.org/funding-experiment-recap/>.

313 Sean T. Larkin, “Trivago Sponsors Webpack for Second Year!,” *Webpack* (blog), Medium, July 9, 2018, <https://medium.com/webpack/trivago-sponsors-webpack-for-second-year-bfe6ca2f0702>.

314 Aboukhadijeh, “Recap of the ‘Funding’ Experiment.”

315 “Elephant Factor,” CHAOSS, n.d.,
<https://chaoss.community/metric-elephant-factor/>.

316 Evan You, Nadia Eghbal, and Mikeal Rogers, “Crowdfunding Open Source (Vue.js) with Evan You,” *Request for Commits*, podcast audio, June 15, 2017, <https://changelog.com/rfc/12>.

317 Antonio Hicks, “Tipping Up 33%, Twitch Viewers Up 21%, Fortnite Dominates—Q118 Streamlabs Report,” *Streamlabs Blog*, April 26, 2018, <https://blog.streamlabs.com/tipping-up-33-twitch-viewers-up-21-fortnite-dominates-q118-streamlabs-report-52f60450af5a>.

318 Jeff Grubb, “Amazon Acquires Twitch: World’s Largest E-Tailer Buys Largest Gameplay-Livestreaming Site,” *VentureBeat*, August 25, 2014, <https://venturebeat.com/2014/08/25/amazon-acquires-twitch-worlds-largest-e-tailer-buys-largest-gameplay-livestreaming-site/>.

319 Ben Thompson, “The Local News Business Model,” *Stratechery*, May 9, 2017, <https://stratechery.com/2017/the-local-news-business-model/>.

320 Tiago Forte, “Why I’m Leaving Medium,” *Praxis*, ForteLabs, April 23, 2018, <https://praxis.fortelabs.co/why-im-leaving-medium/>.

321 Eugene Wei, “Status as a Service (StaaS),” *Remains of the Day*, February 19, 2019, <https://www.eugenewei.com/blog/2019/2/19/status-as-a-service>.

322 Nadia Eghbal, “The Twitch Argument for GitHub Sponsors,” Nadia Eghbal, May 24, 2019, <https://nadiaeghbal.com/github-sponsors>.

323 Wei, “Status as a Service (StaaS).”

324 Jon Gjengset (@jonhoo), “Sadly, I have to close my @Patreon . . .,” Twitter, October 2, 2018, 6:58 p.m.,
<https://twitter.com/Jonhoo/status/1047259437319278592>.

325 Eric S. Raymond, “Load-Bearing Internet People,” *Armed and Dangerous*, June 15, 2019, <http://esr.ibiblio.org/?p=8383>.

326 Laurie Voss (@seldo), “It is very hard to explain npm’s place . . .,” Twitter, April 3, 2018, 6:06 p.m.,
<https://twitter.com/seldo/status/981291962559901696>.

327 Kevin Kelly, “1,000 True Fans,” *The Technium*, March 4, 2008,
<https://kk.org/thetechnium/1000-true-fans/>.

328 Forte, “Why I’m Leaving Medium.”

329 “Hacktoberfest,” Hacktoberfest, accessed April 18, 2020,
<https://hacktoberfest.digitalocean.com/>.

330 T. L. Taylor, *Watch Me Play: Twitch and the Rise of Game Live Streaming* (Princeton, NJ: Princeton University Press, 2018), 260.

331 Ralf Gommers, “Re: [Pandas-dev] Tidelift,” The Pandas-dev Archives, June 11, 2019, <https://mail.python.org/pipermail/pandas-dev/2019-June/000972.html>.

332 “Font Awesome 5,” Kickstarter, March 13, 2018,
<https://www.kickstarter.com/projects/232193852/font-awesome-5>.

333 “Kent Overstreet Is Creating Bcachefs - a Next Generation Linux Filesystem,” Patreon, accessed March 13, 2020,
<https://www.patreon.com/bcachefs>.

334 Na Sun, Patrick Pei-Luen Rau, and Liang Ma, “Understanding Lurkers in Online Communities: A Literature Review,” *Computers in Human Behavior*, no. 38 (September 2014): 110–117,
<https://www.sciencedirect.com/science/article/pii/S0747563214003008>

.

335 “Eran Hammer Is Creating Open Source Software,” Patreon, accessed November 29, 2017, <https://www.patreon.com/eranhammer>.

336 “Support Django,” Django Software Foundation, accessed March 15, 2020, <https://www.djangoproject.com/fundraising/>.

337 Tim Graham, “Django Fellowship Program: A Retrospective,” Django Software Foundation, January 21, 2015,
<https://www.djangoproject.com/weblog/2015/jan/21/django-fellowship-retrospective/>.

338 Matt Holt, “The Realities of Being a FOSS Maintainer,” Caddy Forum, September 3, 2017, <https://caddy.community/t/the-realities-of-being-a-foss-maintainer/2728>.

339 Sindre Sorhus (@sindresorhus), “My Patreon campaign is going well . . . ,” Twitter, March 7, 2019, 12:46 p.m.,
<https://twitter.com/sindresorhus/status/1103713423605432325>

340 “GitHub Sponsors,” GitHub, accessed March 13, 2020,
<https://github.com/sponsors>.

CONCLUSION

341 Kara Swisher, “A Wise Man Leaves Facebook,” *The New York Times*, September 27, 2018,
<https://www.nytimes.com/2018/09/27/opinion/facebook-instagram-systrom.html>.

342 Ian Sullivan, “Re: Things That Happened in November,” email to author, December 20, 2018.

343 Nadia Eghbal, “The Developer’s Dilemma,” Nadia Eghbal, February 8, 2018, <https://nadiaeghbal.com/developers-dilemma>.

344 Shauna Gordon-McKeon, No Subject, email to author, March 22, 2019.

345 Ben Thompson, “Faceless Publishers,” *Stratechery*, May 31, 2017, <https://stratechery.com/2017/the-faceless-publisher/>.

346 Eugene Wei, “Status as a Service (StaaS),” *Remains of the Day*, February 19, 2019, <https://www.eugenewei.com/blog/2019/2/19/status-as-a-service>.

347 Yancey Strickler, “The Dark Forest Theory of the Internet,” *OneZero*, May 20, 2019, <https://onezero.medium.com/the-dark-forest-theory-of-the-internet-7dc3e68a7cb1>.

348 Nick Statt, “Facebook CEO Mark Zuckerberg Says the ‘Future Is Private,’” *The Verge*, April 30, 2019,

<https://www.theverge.com/2019/4/30/18524188/facebook-f8-keynote-mark-zuckerberg-privacy-future-2019>.

349 Sarah Perez, “Twitter Launches Its Controversial ‘Hide Replies’ Feature in the US and Japan,” *TechCrunch*, September 19, 2019,

<https://techcrunch.com/2019/09/19/twitter-launches-its-controversial-hide-replies-feature-in-the-u-s-and-japan/>.

Dieter Bohn, “Twitter Will Put Options to Limit Replies Directly on the Compose Screen,” *The Verge*, January 8, 2020,

<https://www.theverge.com/platform/amp/2020/1/8/21056856/twitter-replies-limit-option-compose-screen-beta-app-features-new-ces-2020>.

350 Josh Constine, “Instagram Launches ‘Stories,’ a Snapchatty Feature for Imperfect Sharing,” *TechCrunch*, August 2, 2016,

<https://techcrunch.com/2016/08/02/instagram-stories/>.

351 Felix Richter, “The Steady Rise of Podcasts,” Statista, March 7, 2019, <https://www.statista.com/chart/10713/podcast-listeners-in-the-united-states/>.

352 Kicks Condor, “Interview about Your Blog?,” email to author, August 29, 2019.

353 “About the Creator Account on Instagram,” Instagram Help Center, n.d., <https://help.instagram.com/1158274571010880?helpref=related>.

354 Janko Roettgers, “Instagram to Start Hiding Like Counts in the U.S.,” *Variety*, November 8, 2019,

<https://variety.com/2019/digital/news/instagram-likes-like-counts-hidden-1203399222/>.

Sarah Perez, “YouTube Confirms a Test Where the Comments Are Hidden by Default,” *TechCrunch*, June 21, 2019,

<https://techcrunch.com/2019/06/21/youtube-confirms-a-test-where-the-comments-are-hidden-by-default/>.

355 Taylor Swift, “30 Things I Learned before Turning 30,” *ELLE*, March 6, 2019,

<https://www.elle.com/culture/celebrities/a26628467/taylor-swift-30th-birthday-lessons/>.

356 Tavi Gevinson, “Who Would I Be Without Instagram? An Investigation,” *The Cut*, September 16, 2019,

<https://www.thecut.com/2019/09/who-would-tavi-gevinson-be-without-instagram.html>.

357 “Dear GitHub,” Dear-Github Code, GitHub, January 10, 2016,

<https://github.com/dear-github/dear-github>.

358 Jake Boxer, “Add Reactions to Pull Requests, Issues, and Comments,” *The GitHub Blog*, March 10, 2016,

<https://github.blog/2016-03-10-add-reactions-to-pull-requests-issues-and-comments/>.

359 Edward T. Hall, *The Hidden Dimension*, Later printing ed. (New York: Anchor Books, 1990), 106.

360 Tim Ferriss, “Why I’m Stopping the Fan-Supported Podcast Experiment,” *Tim Ferriss’s 4-Hour Workweek and Lifestyle Design Blog*, July 11, 2019, <https://tim.blog/2019/07/11/why-im-stopping-the-fan-supported-podcast-experiment/>.

361 Taylor Wofford, “Fuck You and Die: An Oral History of Something Awful,” *Vice*, April 5, 2017, https://www.vice.com/amp/en_us/article/nzg4yw/fuck-you-and-die-an-oral-history-of-something-awful.

362 Nick Szabo, “Micropayments and Mental Transaction Costs,” Nakamoto Institute, n.d., <https://nakamotoinstitute.org/static/docs/micropayments-and-mental-transaction-costs.pdf>.

363 Tim Carmody, “Statement of Purpose,” *Amazon Chronicles*, January 27, 2019, <https://amazonchronicles.substack.com/p/statement-of-purpose>.

364 Tim Carmody, “Unlocking the Commons: Or, the Psychoeconomics of Patronage,” *Kottke.org*, December 15, 2017, <https://kottke.org/17/12/unlocking-the-commons-or-the-psychoeconomics-of-patronage>

365 Matthew Butterick, “To Pay or Not to Pay: How I Profited from Gentle Shame,” *Butterick’s Practical Typography*, August 5, 2016, <https://practicaltypography.com/to-pay-or-not-to-pay.html>.

366 Damon Kiesow, “Journalism’s Dunbar Number: Audience Scales, Community Does Not,” Local News Lab, March 4, 2019,

<https://localnewslab.org/2019/03/04/journalisms-dunbar-number-audience-scales-community-does-not/>.

367 Alex Kantrowitz, “Paid Email Newsletters Are Proving Themselves as a Meaningful Revenue Generator for Writers,” BuzzFeed, April 29, 2019,

<https://www.buzzfeed.com/alexkantrowitz/writers-have-been-trying-to-support-online-themselves-for>

368 Kevin Draper, “Why The Athletic Wants to Pillage Newspapers,” *The New York Times*, October 23, 2017,

<https://www.nytimes.com/2017/10/23/sports/the-athletic-newspapers.html>.

369 David Bauder and David A. Lieb, “Decline in Readers, Ads Leads Hundreds of Newspapers to Fold,” Associated Press, March 11, 2019,
<https://apnews.com/0c59cf4a09114238af55fe18e32bc454>

ABOUT THE AUTHOR

Nadia Eghbal is a writer and researcher who explores how the internet enables individual creators. From 2015 to 2019, she focused on the production of open source software, working independently and at GitHub to improve the open source developer experience. She is the author of “Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure,” published by the Ford Foundation, in which she argued that open source code is a form of public infrastructure that requires maintenance.