



How Open Source Ate Software

Understand the Open Source
Movement and So Much More

Gordon Haff

Apress®

How Open Source Ate Software

**Understand the Open Source
Movement and So Much More**

Gordon Haff

Apress®

How Open Source Ate Software

Gordon Haff
Lancaster, Massachusetts, USA

ISBN-13 (pbk): 978-1-4842-3893-6
<https://doi.org/10.1007/978-1-4842-3894-3>

ISBN-13 (electronic): 978-1-4842-3894-3

Library of Congress Control Number: 2018953693

Copyright © 2018 by Gordon Haff

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: Louise Corrigan

Development Editor: James Markham

Coordinating Editor: Nancy Chen

Cover designed by eStudioCalamar

Cover image by Opensource.com, CC BY-SA 4.0

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484238936. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To my dad.

Table of Contents

About the Author	xiii
Acknowledgments	xv
Introduction	xvii
Chapter 1: The Beginnings of Free and Open Source Software	1
In the Beginning	1
Ah, Unix	2
No More Free Lunches?.....	3
PCs Were a Different Culture	5
Breaking Community	6
Free Software Enters the Fray.....	7
Establishing the Foundations of Free	8
Fragmented Hardware and Software	9
Vertical Silos Everywhere	9
Silos Turn On Their Side.....	10
Which Mass-Market Operating System Would Prevail?.....	11
Microsoft Swings for the Fences.....	11
Windows NT Poised to Take It All.....	12
The Internet Enters the Mainstream	13
From Scale-Up to Scale-Out.....	13
Internet Servers Needed an Operating System	14
Enter Linux	14
A New *nix	15
Linux Grows in Popularity	15
Eclipsing Unix	16
Open Source Accelerates	16
A New Enterprise IT Model	17
Born on the Web	17

TABLE OF CONTENTS

Build or Buy?	18
Disrupting the Status Quo	18
From Disruption to Where Innovation Happens	20
The Rise of Ecosystems.....	21
Breaking Up Monoliths	21
Linux and Open Source Had Arrived	22
Chapter 2: From “Free” to “Open Source”	25
Words Can Matter	25
Why Free.....	26
The Coining of “Open Source”	26
Pragmatism and Commercialism	27
How Open Source Licensing Works	28
Do You Have to Give Back or Not?	28
Protecting the Commons	29
Seeing Through the Copyleft Mire	30
Permissive Licenses Gain.....	30
Driving Participation Is the Key	31
Maintaining Open Source Compliance	32
Putting Controls in Place	32
What Are Your Policies?	32
An Ongoing Process	33
Projects versus Products	33
Upstream and Downstream.....	33
Projects and Products Depend on Each Other.....	35
What Support Means	35
Reducing Risk.....	36
The Intersection of Security and Risk.....	36
Securing Open Source	37
Business as Usual: Patches and Advice	37
Does Code Help the Bad Guys?	38
Or Is “Many Eyes” the Secret Sauce?	38
Thinking Differently About Risk	39

Participating in Open Source Projects	40
Starting an Open Source Project	41
Doubling Down an Existing Open Source Project	43
Creating an Open Source Program Office.....	45
The Water Is Fine	47
Chapter 3: Open Source Development Model	49
Open Source Is About Development.....	49
Central versus Distributed Control.....	50
Differing Open Source Approaches.....	50
A Caveat	51
Models for Governing Projects.....	51
Who Decides?.....	52
What Are the Principles?	56
Best Practices for Setting Goals	57
Who Is in the Community?.....	59
Why You Should Think About More Than Coders.....	61
Users Get Involved.....	62
Users Become Contributors.....	62
How to Encourage New Contributors	63
Holding Onto Control: An Anti-Pattern	63
Reducing the Friction of Tools	64
Mentoring	65
The Importance of Culture	66
Steps to Maintain a Community.....	66
Quick Responses	67
Documentation: An Easy On-Ramp	67
Modular Beats Monoliths.....	68
Communicate, Communicate, Communicate	68
The Limits of Being Together	69
Best Practices for Distributed Teams	69
It's About People.....	70
The Limits of Virtual.....	70

TABLE OF CONTENTS

Determine If You're Successful	72
Measuring Something Changes It	72
What Actually Matters?	73
Quantity May Not Lead to Quality	73
What Does the Number Mean?	74
Horses for Courses	75
Understand Softer Aspects of Community	75
Back to the Bazaar	76
It's a Bit of a Free-for-All	77
Open Source Is Duplicative	77
Community Makes It Work	78
Why the Development Model Matters	79
Chapter 4: Open Source's Connection to the Past	81
The Machine That Drives Open Source	81
Innovation	82
Innovation Through Collective Invention	82
The Economics of Open	83
The Advantages of Collaborative Innovation	84
How Knowledge Gets Shared	85
Collaboration and Communication	85
The Limits of Communication	86
How Communication Affects Software Structure	87
Modularity Is Generally Better	88
How Contributors Interact in Open Source	89
Participation	90
How Participants Get Started	90
Onboarding and Mentoring	91
Motivation	92
The Types of Motivators	92
Research into Open Source Motivations	93
Intrinsic Motivators	93
Internalized Extrinsic Motivators	94
Extrinsic Motivators	95

Measurement	96
Why Measure?	96
Measurements Affect Behaviors.....	96
The Limits of Direct Measures.....	97
Measurements at Cross-Purposes	97
DevOps and Metrics	98
Throughput Measures	99
Stability Measures.....	99
Understanding Community Health.....	100
Shining More Light on Culture	101
Diversity and Inclusion	101
Project Growth-Maturity-Decline.....	102
Risk.....	102
Value.....	102
A Broader View of Health.....	103
Reflecting and Informing	104
Chapter 5: Business Models	105
How Can You Sell Something That You Give Away?	105
Is There an “Open Source Business Model”?	106
Categories of Business Models	107
Getting the Balance Right.....	107
Building the Funnel with Free.....	108
Open Core versus Open Source	108
Are You Taking Advantage of Open Source Development?	109
Subscriptions and Support	110
The Rise of the Independent Software Vendor	111
Open Source Support Arrives	111
Linux Distributions Appear	112
Subscriptions: Beyond Support	113
Focusing on Core Competencies	113
Aligning Incentives with Subscriptions	114

TABLE OF CONTENTS

From Competition to Coopetition	115
Coopetition Gets Coined	115
Why Coopetition Has Grown	116
Open Source: Beneficiary and Catalyst	117
Coopetition and Standards	118
The Need for Speed	119
From Physical to Virtual.....	119
The Consumerization of IT	119
The Rise of DevOps	120
The DevOps Origin Story.....	120
DevOps: Extending Beyond Agile	122
Abstractions to Separate Concerns	122
Site Reliability Engineers.....	123
Manufacturing Analogs.....	124
Open Source and DevOps	124
Platforms and Tooling.....	124
Process.....	125
Pervasive Open Source	130
Chapter 6: The Flip Side of Open Source	131
A Positive Feedback Loop	131
Projects	132
Solutions and Products.....	132
Value.....	133
The IT Industry Has Changed	133
The Rise of the “Cloud”	134
Why the Cloud Matters to Open Source.....	136
Out at the Edge.....	137
Where’s the Money?	138
What Users Want.....	139
The New Bundles.....	139
Users Want Convenience	141
Hiding Complexity.....	142

It's Not the End.....	143
IT Is Hybrid.....	143
Centers of Gravity Shift.....	144
The Development Model Is Effective	145
Ecosystems Matter	146
It's Not Just About the Code	149
Chapter 7: Open Source Opportunities and Challenges	151
Opening Data	151
Value from Data	152
Transparency Through Data.....	153
Ownership of Data.....	155
Opening Information	156
The Read/Write Web	156
Wikipedia.....	157
Independent Contribution	158
Opening Education	158
Precursors	159
MIT OpenCourseWare	159
MOOCs.....	160
Collaboration versus Consumption.....	162
Opening Hardware	163
Ham Radio	163
A Shift from Making.....	164
The New Makers.....	166
Opening Culture in Organizations	168
Why Do Organizations Exist Anyway?.....	168
Open Organizations	169
Concluding Thoughts	172
Index.....	175

About the Author



Gordon Haff is a technology evangelist for Red Hat, the leading provider of commercial open source software. He is a frequent speaker at customer and industry events. He writes for a variety of publications including *The Enterprisers Project*, *opensource.com*, *Connections*, and *TechTarget*. His *Cloudy Chat* podcast includes interviews with a wide range of industry experts. He also works on strategy for Red Hat's hybrid cloud portfolio and other emerging technology areas such as IoT, Blockchain, AI, and DevOps.

Prior to Red Hat, as an IT industry analyst, Gordon wrote hundreds of research notes, was frequently quoted in publications such as *The New York Times* on a wide range of IT topics, and has advised clients on product and marketing strategies. Earlier in his career, he was responsible for bringing a wide range of computer systems, from minicomputers to large Unix servers, to market while at Data General.

He lives west of Boston, Massachusetts, in apple orchard country and is an avid hiker, skier, sea kayaker, and photographer. He can be found on Twitter as @ghaff and by email at gordon@alum.mit.edu. His website and blog are at <http://www.bitmasons.com>.

Gordon has engineering degrees from MIT and Dartmouth and an MBA from Cornell's Johnson School.

Acknowledgments

I'd like to thank my employer, Red Hat, for its support in writing this book. Red Hat also contributes to many of the open source projects that I discuss throughout these pages.

Over the years, many Red Hatters have contributed to and helped to develop many of the ideas and concepts explored here, as well as building the business practices that have demonstrated how open source can anchor effective business models. A number of my colleagues have also provided specific feedback and contributed content including William Henry, David Neary, Joe Brockmeier, Ross Turk, Robyn Bergeron, Deb Bryant, Thomas Cameron, Diane Mueller, and Gina Likins, among others.

The Linux Foundation executive, event, and public relations teams contributed through public resources, interviews, and the many fertile discussions that their events have been a catalyst for.

I'd also like to thank my editors at Apress—Nancy Chen, Louise Corrigan, and James Markham for their efforts in bringing this manuscript to the page—as well as managing director Welmoed Spahr for agreeing to take on this project.

Finally, I express my appreciation to all the interview subjects, researchers, other authors, and editors without whom this book would not be possible.

Introduction

Information wants to be free. Users are in control. I have rights to the code I depend on. That's one way to view free and open source software.

Those are important things. Software freedoms matter.

But open source also has practical applications relating to how software is developed and used. It's part of a broader evolution in how individuals and organizations cooperatively innovate.

Open source software arose in a specific time and place when and where norms that had once prioritized sharing were giving way to a more proprietary and closed industry computer industry. Open source software helped to change that and, in so doing, it illuminated processes that were better for innovation and not just preferable for mostly abstract reasons relating to software freedom.

Open source has become an integral part of a world transitioning from traditional industrial structures and embedding software into more and more of its fabric.

This book takes you through that history. It explains how open source works from community, development, and business perspectives through the eyes and words of both practitioners and researchers. It examines the challenges facing open source given that some of the conditions that gave rise to it no longer apply. It takes you through the many ways that open source approaches are taking root in data, hardware, education, and organizations more broadly.

Not everything is transparent and open, nor should it be. Business models depend on unique capabilities and knowledge. However, open source software both reflects and informs trends and practices that are becoming increasingly important and pervasive in business and elsewhere. Understand how open source came to be and how it works, and you'll be closer to understanding how to innovate in the modern world.

CHAPTER 1

The Beginnings of Free and Open Source Software

Origin stories are often messy. What you are about to read is no exception.

The story of open source software is a maze of twisty streams feeding into each other and into the main channel. They're part and parcel with the history of the Unix operating system, which itself is famously convoluted. In this chapter, we'll take a look at open source's humble beginnings.

In the Beginning

Going even further back, the sharing of human-readable source code was widespread in the early days of computing. A lot of computer development took place at universities and in corporate research departments like AT&T's Bell Labs. They had long-established traditions of openness and collaboration, with the result that even when code wasn't formally placed into the public domain, it was widely shared.

Computer companies shipping software with their systems also often included the source code. Users frequently had to modify the software themselves so that it would support new hardware or because they needed to add a new feature. The attitude of many vendors at the time was that software was something you needed to use the hardware, but it wasn't really a distinct thing to be sold.

Indeed, users of computer systems often had to write their own software in the early days. The first operating system for the IBM 704 computer (Figure 1-1), the GM-NAA I/O input/output system, was written in 1956 by Robert L. Patrick of General Motors Research and Owen Mock of North American Aviation.



Figure 1-1. IBM 704 at NASA Langley in 1957. Source: Public domain.

The culture of sharing code—at least in some circles—was still strong going into the late 1970s when John Lions of the University of New South Wales in Australia annotated the Sixth Edition source code of the Unix operating system. Copies of this “Lions’ Commentary” circulated widely among university computer science departments and elsewhere. This sort of casual and informal sharing was the norm of the time even when it wasn’t really technically allowed by the code’s owner, in this case AT&T.

Ah, Unix

The idea of modifying software to run on new or different hardware gained momentum around Unix. Unix was rewritten in 1973–1974 (for its V4 release) in C, a programming language newly developed by Dennis Ritchie at Bell Labs. Using what was, by the

standards of the time, a high-level programming language for this purpose, while not absolutely unique, was nonetheless an unusual, new, and even controversial approach.

More typical would have been to use the assembly language specific to a given machine's architecture. Because of the close correspondence between the assembler and an architecture's machine code instructions (which execute directly on the hardware), the assembler was extremely efficient, if challenging and time-consuming to write well. And efficiency was important at a time when there wasn't a lot of spare computer performance to be wasted.

However, as rewritten in C, Unix could be modified to work on other machines relatively easily; that is, it was "portable." This was truly unusual. The norm of the day was to write a new operating system and set of supporting systems and application software for each new hardware platform.

Of course, to make those modifications, you needed the source code.

AT&T was willing to supply this for several reasons. One was especially important. After the Sixth Edition was released in 1975, AT&T began licensing Unix to universities and commercial firms, as well as the United States government. But the licenses did not include any support or bug fixes, because to do so would have been "pursuing software as a business," which AT&T did not believe that it had the right to do under the terms of the agreement by which it operated as a regulated telephone service monopoly. The source code let licensees make their own fixes and "port" Unix to new systems.

However, in the early 1980s, laid-back attitudes toward sharing software source code started to come to an end throughout the industry.

No More Free Lunches?

In AT&T's specific case, 1982 was the year it entered into a consent decree with the Federal Trade Commission providing for the spin-off of the regional Bell operating companies (Figure 1-2). Among other things, the decree freed AT&T to enter the computer industry. Shortly thereafter, AT&T commenced development of a commercial version of Unix.

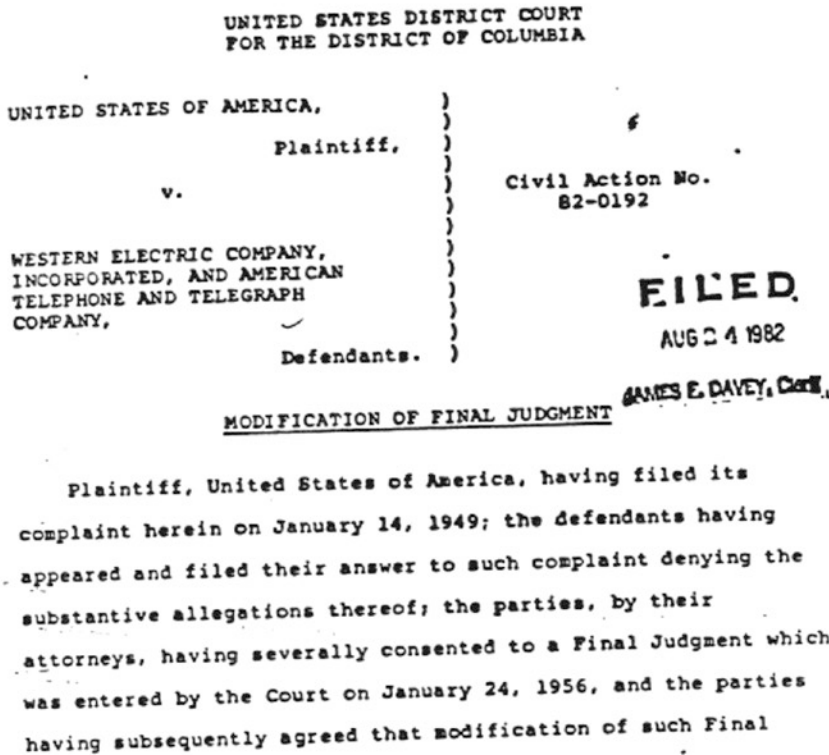


Figure 1-2. AT&T entered into a consent decree in 1982 that allowed it to pursue software as a business. This led to the increasing commercialization of Unix.
Source: Public domain.

This would lead, over the course of about the next decade, to the messy “Unix Wars” as AT&T Unix licensees developed and shipped proprietary Unix versions that were all incompatible with each other to greater or lesser degrees. It’s an *extremely* complicated and multi-threaded history. It’s also not all that relevant to how open source has evolved other than to note that it created vertical silos that weren’t all that different from the minicomputers and mainframes that they replaced. The new boss looked a lot like the old boss.

During the same period, AT&T—in the guise of its new Unix System Laboratories subsidiary—got into a legal fight with the University of California, Berkeley over their derivative (a.k.a. “fork”) of Unix, the Berkeley System Distribution (BSD). Specifically, it was a squabble with the Computer Systems Research Group (CSRG) at Berkeley, but I’ll just refer to the university as a whole here.

Berkeley had been one of AT&T's educational licensees. Over time, it modified and added features to its licensed version of Unix and, in 1978, began shipping those add-ons as BSD. Over time it added significant features, involving the outright re-architecting and rewriting of many key subsystems, and the addition of many wholly new components. As a result of its extensive changes and improvements, BSD was increasingly seen as an entirely new, even better, strain of Unix; many AT&T licensees would end up incorporating significant amounts of BSD code into their own Unix versions. (Which contributed further to the Unix wars as different companies favored the AT&T strain or the Berkeley strain in their products.)

Berkeley continued developing BSD to incrementally replace most of the standard Unix utilities that were still under AT&T licenses. This eventually culminated in the June 1991 release of Net/2, a nearly complete operating system that was ostensibly freely redistributable. This in turn led to AT&T suing Berkeley for copyright infringement.

Suffice it to say that the commercialization of Unix, which had been the hub around which much source code sharing had taken place, helped lead to a more balkanized and closed Unix environment.

PCs Were a Different Culture

But the sharing ethos was also eroding more broadly.

During the 1980s, the personal computer space was increasingly dominated by the IBM PC and its clones running a Microsoft operating system. Nothing that looked much like open source developed there to a significant degree. In part, this probably reflected the fact that the relatively standardized system architecture of the PC made the portability benefits of having source code less important.

Furthermore, most of the tools needed to develop software weren't included when someone bought a PC and the bill for those could add up quickly. A bare-bones BASIC programming language interpreter was included with Microsoft's DOS operating system, but that was seen as hopelessly outdated for serious programming, even by the not-so-demanding standards of the time. When Borland's more modern Turbo Pascal debuted in 1984 for only 50 dollars, it was a radical innovation given that typical programming language packages went for *hundreds* of dollars. Programming libraries and other resources—including information that was mostly locked up in books, magazine, and other offline dead tree sources—added to the bill. Making a few changes to a piece of software was not for the casual hobbyist.

People did program for the IBM PC, of course, and over time a very healthy community of freeware and shareware software authors came into being.

I was one of them.

Shareware, at least as the term was generally used at the time, meant try-before-you-buy software. Remember, this is a time when boxed software sold at retail could go for hundreds of dollars with no guarantee that it would even work properly on your computer. And good luck returning it.

The main software I wrote was a little DOS file manager, 35KB of assembler, called Directory Freedom, derived from some assembly code listings in *PC Magazine* and another developer's work. It never made a huge amount of money, but it had its fan base and I still get emails about it from time to time. I also wrote and uploaded to the local subscription bulletin board system (BBS) various utility programs that I originally wrote for my own use.

But distributing source code was never a particularly big thing.

Breaking Community

Similar commercializing dynamics were playing out in other places. The MIT Artificial Intelligence (AI) Lab, celebrated by Steven Levy in *Hackers* as a “pure hacker paradise, the Tech Square monastery where one lived to hack, and hacked to live,” was changing. Here, it was Lisp that was commercializing.

The Lisp programming language was the workhorse of artificial intelligence research, but it required so many hardware resources that it didn't run well on the ordinary computers of the day. As a result, for close to a decade, members of the AI Lab experimented with systems that were optimized to run Lisp. By 1979, that work had progressed to the point where commercialization looked like a valid option.

Eventually two companies, Symbolics and Lisp Machines Inc., would be formed. But it ended up as a messy and acrimonious process that led to much reduced open collaboration and widespread departures from the Lab.

Richard Stallman was one member of the AI Lab who did not head off to greener corporate Lisp pastures but nonetheless felt greatly affected by the splintering of the Lab community. Stallman had previously written the widelyused Emacs editing program. With Emacs, as Glyn Moody writes in *Rebel Code*, “Stallman established an ‘informal rule that anyone making improvements had to send them back’ to him.”

His experiences with the effects of proprietary code in the Symbolics versus Lisp Machines Inc. war led him to decide to develop a free and portable operating system, given that he had seen a lack of sharing stifling the formation of software communities. In another widely told story about Stallman's genesis as a free software advocate, he was refused access to the source code for the software of a newly installed laser printer, the Xerox 9700, which kept him from modifying the software to send notifications as he had done with the Lab's previous laser printer.

Free Software Enters the Fray

In 1983, Richard Stallman announced on Usenet, the Internet's (actually its predecessor ARPANET at the time) newsgroup service, that "Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (for Gnu's Not Unix), and give it away free to everyone who can use it"; the logo can be seen in Figure 1-3.



Figure 1-3. Stallman's Free Software Foundation and GNU project are generally taken as the beginning of free and open source software as a coherent movement. Source: Victor Siame vcopovi@wanadoo.fr under Free Art License.

As justification, he went on to write that “I consider that the golden rule requires that if I like a program I must share it with other people who like it. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. So that I can continue to use computers without violating my principles, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free.”

It was to be based on the Unix model, which is to say that it was to consist of modular components like utilities and the C language compiler that’s needed to build a working system. The project began in 1984. To this day, there is in fact no “GNU operating system” in that the GNU Hurd operating system kernel has never been completed. Without a kernel, there’s no way to run utilities, applications, or other software as they have no way to communicate with the hardware.

However, Stallman did complete many other components of his operating system. These included, critically, the parts needed to build a functioning operating system from source code and to perform fundamental system tasks from the command line. It’s a hallmark of Unix that its design is very modular. As a result, it’s entirely feasible to modify and adapt parts of Unix without wholesale replacing the whole thing at one time. (A fact that would be central to the later development of Linux.)

Establishing the Foundations of Free

However, equally important from the perspective of open source’s origins was the GNU Manifesto that followed in 1985, the Free Software Definition in 1986, and the GNU Public License (GPL) in 1989, which formalized principles for preventing restrictions on the freedoms that define free software.

The GPL requires that if you distribute a program covered by the GPL in binary, that is, machine-readable form, whether in original or modified form, you must also make the human-readable source code available. In this way, you can build on both the original program and the improvements of others but, if you yourself make changes and distribute them, you also have to make those changes available for others to use. It’s what’s known as a “copyleft” or reciprocal license because of this mutual obligation.

Free and open source software was still in its infancy in the late 1980s. (Indeed, the “open source” term hadn’t even been coined yet.) Linux was not yet born. BSD Unix would soon be embroiled in a lawsuit with AT&T. The Internet was not yet fully commercialized. But, especially with the benefit of hindsight, we can start to discern patterns that would become important: collaboration, giving back, and frameworks that help people to know the rules and work together appropriately.

But it was the Internet boom of the 1990s that would really put Linux and open source on the map even if this phase of open source would turn out to be just the first act of an ultimately more important story. This is the backdrop against which open source would rise in prominence while the computer hardware and software landscape shifted radically.

Fragmented Hardware and Software

Turn the clock back to 1991. A Finnish university student by the name of Linus Torvalds posted in a Usenet newsgroup that he was starting to work on a free operating system in the Unix mold as a hobby. Many parts of Stallman's initial GNU Project were complete. In sunny California, Berkeley has shipped the first version of its Unix to be freely distributable.

Free software had clearly arrived. It just wasn't a very important part of the computing landscape yet.

Vertical Silos Everywhere

It was a very fragmented computing landscape. The Unix market was embroiled in internecine proprietary system warfare. Many other types of proprietary computer companies were also still around—if often past their prime.

The most prominent were the “Route 128” Massachusetts minicomputer companies, so called because many were located on or near the highway by that name, which partially encircled the adjacent cities of Boston and Cambridge on the northeast coast of the United States. However, there were also many other vendors who built and sold systems for both commercial and scientific computing. Most used their own hardware designs from the chips up through disk drives, tape drives, terminals, and more. If you bought a Data General computer, you also bought memory, reel-to-reel tape drives, disk drives, and even cabinets from either the same company or a small number of knock-off add-on suppliers.

Their software was mostly one-off as well. A typical company would write its own operating system (or several different ones) in addition to databases, programming languages, utilities, and office applications. When I worked at Data General during this period, we had about five different non-Unix minicomputer operating systems plus a couple of different versions of Unix.

Many of these companies were themselves increasingly considering a wholesale shift to their own versions of Unix. But it was mostly to yet another customized hardware and Unix operating system variant.

Most computer systems were still large and expensive in those days. “Big Iron” was the common slang term. The analysis and comparison of their complicated and varied architectures filled many an analyst’s report.

Even “small business” or “departmental” servers, as systems that didn’t require the special conditions of the “glass room” datacenter were often called, could run into the tens of thousands of dollars.

Silos Turn On Their Side

However, personal computers were increasingly starting to be stuck under desks and used for less strenuous tasks. Software from Novell called NetWare, which specialized in handling common tasks like printing or storing files, was one common option for such systems. There were also mass-market versions of Unix. The most common came from a company called Santa Cruz Operation that had gotten into the Unix business by buying an AT&T-licensed variant called Xenix from Microsoft. Many years later, Santa Cruz Operation—or more accurately a descendent of them using the name SCO—would instigate a series of multiyear lawsuits related to Linux that would pull in IBM and others.

More broadly, there was a pervasive sea change going on in the computer systems landscape. As recounted by semiconductor maker Intel CEO Andy Grove in *Only the Paranoid Survive*, a fundamental transformation happened in the computer industry during the 1990s. As we’ve seen, the historical computer industry was organized in vertical stacks. Those vertical stacks were increasingly being rotated into a more horizontal structure.

It wasn’t a pure transformation; there were (and are) still proprietary processors, servers, and operating systems.

But more and more of the market was shifting toward a model in which a system vendor would buy the computer’s central processing unit from Intel, a variety of other largely standardized chips and components from other suppliers, and an operating system and other software from still other companies. They’d then sell these “industry standard” servers through a combination of direct sales, mail order, and retail.

During this period, Advanced Micro Devices (AMD) was also producing compatible x86 architecture processors under license from Intel although the two companies would be embroiled in a variety of contractual disputes over time. AMD would later enjoy a

short period of some success against Intel with its Opteron processors but has largely remained in Intel's shadow.

The PC model was taking over the server space.

Grove described this as the 10X force of the personal computer. The tight integration of the old model might be lacking. But in exchange for a certain amount of do-it-yourself to get everything working together, for a few thousand dollars you got capabilities that increasingly rivaled those of engineering workstations you might have paid *tens* of thousands to one of the proprietary Unix vendors to obtain.

Which Mass-Market Operating System Would Prevail?

With the increasing dominance of x86 established, there was now just a need to determine which operating system would similarly dominate this horizontal stack. There was also the question of who would dominate important aspects of the horizontal platform more broadly such as the runtimes for applications, databases, and areas that were just starting to become important like web servers. But those were less immediately pressing concerns.

The answer wasn't immediately obvious. Microsoft's popular MS-DOS and initial versions of Windows were designed for single-user PCs. They couldn't support multiple users like Unix could and therefore weren't suitable for business users who needed systems that would let them easily share data and other resources. Novell NetWare was one multiuser alternative that was very good at what it did—sharing files and printers—but it wasn't a general purpose operating system. And, while there were Unix options for small systems, they weren't really mass market.

Microsoft Swings for the Fences

Microsoft decided to build on its desktop PC domination to similarly dominate servers.

Microsoft's initial foray into a next-generation operating system ended poorly. IBM and Microsoft signed a "Joint Development Agreement" in August 1985 to develop what would later become OS/2. However, especially after Windows 3.0 became a success on desktop PCs in 1990, the two companies increasingly couldn't square their technical and cultural differences. For example, IBM was primarily focused on selling OS/2 to run on its own systems—naming its high-profile PC lineup PS/2 may have been a clue—whereas Microsoft wanted OS/2 to run on a wide range of hardware from many vendors.

As a result, Microsoft had started to work in parallel on a re-architected version of Windows. CEO Bill Gates hired Dave Cutler in 1988. Cutler had led the team that created the VMS operating system for Digital's VAX computer line among other Digital operating systems. Cutler's push to develop this new operating system is well-chronicled in G. Pascal Zachary's *Show Stopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft* (Free Press, 1994) in which the author describes him as a brilliant and, at times, brutally aggressive chief architect.

Cutler had a low opinion of OS/2. He also had a low opinion of Unix. In *Show Stopper!* a team member is quoted as saying "He thinks Unix is a junk operating system designed by a committee of Ph.D.s. There's never been one mind behind the whole thing, and it shows, so he's always been out to get Unix. But this is the first time he's had the chance."

As a result, Cutler undertook the design of a new operating system that would be named Windows NT upon its release in 1993.

IBM continued to work on OS/2 by itself, but it failed to attract application developers, was never a success, and was eventually discontinued. This Microsoft success at the expense of IBM was an early-on example of the growing importance of developers and developer mindshare, a trend that Bill Gates and Microsoft had long recognized and played to considerable advantage. And it would later become a critical factor in the success of open source communities.

Windows NT Poised to Take It All

Windows NT on Intel was a breakout product. Indeed, Microsoft and Intel became so successful and dominant that the "Wintel" term was increasingly used to refer to the most dominant type of system in the entire industry. By the mid-1990s, Unix was in decline, as were other operating systems such as NetWare.

Windows NT was mostly capturing share from Unix on smaller servers, but many thought they saw a future in which Wintel was everywhere. Unix systems vendors, with the notable exception of Sun Microsystems under combative CEO Scott McNealy, started to place side bets on Windows NT. There was a sense of inevitability in many circles.

The irony was that, absent Windows NT, Unix would likely have conquered all. Jeff Atwood wrote that "The world has devolved into Unix and NT camps exclusively. Without NT, I think we'd all be running Unix at this point, for better or worse. It certainly happened to Apple; their next-generation Copland OS never even got off the ground. And now they're using OS X which is based on Unix."

Unix might still have remained the operating system of choice for large systems with many processors; Windows NT was initially optimized for smaller systems. But it was easy to see that Windows NT was fully capable of scaling up and had been architected by Cutler to be able to serve as a Unix replacement. Once it got there, it was going to be very difficult not to rally around something that had become an industry standard just as Intel's x86 processor line had. Products selling in large volume have lower unit costs and find it far easier to establish partnerships and integrations up and down the new stack with its horizontal layers.

The Internet Enters the Mainstream

But wait. It wasn't quite "Game over man." A couple other things were happening by now. The Internet was taking off and the first versions of Linux had been released.

By 1990, the Internet had been in existence, in some form, for a couple of decades. It originated with work commissioned by the US Defense Advanced Research Projects Agency (DARPA) in the 1960s to build fault-tolerant communication with computer networks. However, you probably hadn't heard of it unless you were a researcher at one of the handful of institutions connected to the early network. The Internet emerged from this obscurity in the 1990s for a variety of reasons, not least of which was the invention of the World Wide Web—which for many people today is synonymous with the Internet—by English scientist Tim Berners-Lee while working at CERN.

From Scale-Up to Scale-Out

The great Internet build-out of the late 1990s lifted many boats, including the vendors of high-end expensive hardware and software. The quartet of Sun, networking specialist Cisco, storage disk array vendor EMC, and database giant Oracle was nicknamed the "four horsemen of the Internet." It seemed as if every venture capital-backed startup needed to write a big check to those four.

However, a lot of Internet infrastructure (think web servers)—as well as high-performance scientific computing clusters—ran on large numbers of smaller systems instead. Unix vendors, most notably Sun, were happy to sell their own smaller boxes for these purposes, but those typically carried a lot higher price tag than was the norm for "industry standard" hardware and software.

The use of more and smaller servers was part of a broader industry shift in focus from “scale-up” computing to “scale-out.” Distributed computing was driven by the maturation of the client/server and network/distributed computing styles that first emerged and gained popularity in the 1980s. This seemed like something that played to the Microsoft wheelhouse.

Internet Servers Needed an Operating System

Yet, Windows NT wasn’t really ideal for these applications either. Yes, it was rapidly taking share from Unix in markets like small business and replicated sites in larger businesses—to the point where a vendor like Santa Cruz selling primarily into those markets was facing big losses. However, network infrastructure and scientific computing roles had mostly favored Unix historically for both reasons of custom and technology. For example, the modular mix-and-match nature of Unix had long made it popular with tinkerers and do-it-yourselfers. Transitioning to Windows NT was therefore neither natural nor easy.

BSD Unix was one obvious alternative. It did make inroads but only limited ones. The reasons are complicated and not totally clear even with the benefit of hindsight. Lingering effects of the litigation with AT&T. Licensing that didn’t compel contributing back like the GPL does. A more centralized community that was less welcoming to outside contributions. I’ll return to some of these later, but, in any case, BSD didn’t end up having a big effect on the software landscape.

Enter Linux

Into this gap stepped Linux paired with other open source software such as GNU, the Apache web server, and many other types of software over time.

Recall our Finnish university student Linus Torvalds. Working on and inspired by MINIX, a version of Unix initially created by Andrew Tanenbaum for educational purposes, Torvalds began writing an operating system kernel.

The kernel is the core of a computer’s operating system. Indeed, some purists argue that a kernel *is* the operating system with everything else part of a broader operating environment. In any case, it is usually one of the first programs loaded when a computer starts up. A kernel connects application software to the hardware of a computer and generally abstracts the business of managing the system hardware from “userspace” things that someone trying to use the computer to do something cares about.

A New *nix

Linux is a member of the Unix-like (or sometimes “*nix”) family of operating systems. The distinction between Unix and Unix-like is complicated, unclear, and, frankly, not very interesting. Originally, the term “Unix” meant a particular product developed by AT&T. Later it extended to AT&T licensees. Today, The Open Group owns the Unix trademark, but the term has arguably become generic with no meaningful distinction between operating systems that are indisputably a true Unix and those that are better described as Unix-like for reasons of trademark or technical features.

Torvalds announced his then-hobby project in 1991 to the broader world in a Usenet posting to the newsgroup comp.os.minix. He soon released version 0.01, which was mostly just the kernel.

By the next year, Torvalds had relicensed Linux under the GPL. Others had created the first Linux distributions to simplify installing it and to start packaging together the many components needed to use an operating system. By 1993, over 100 developers were working on the Linux kernel; among other things they adapted it to the GNU environment. The year 1994 saw version 1.0 of Linux, the addition of a graphical user interface from the XFree86 project, and commercial Linux distributions from Red Hat and SUSE.

Linux Grows in Popularity

Initially, Linux was popular especially in university and computing research organizations, much as Unix itself had been starting in the mid-1970s, and Solaris from the mid-1980s. Linux also started finding its way into many network infrastructure roles for file and print sharing, Web and FTP serving, and similar tasks.

You could download it for free or buy it on a disk for cheap. It’s worth being explicit about what “cheap” meant in this context. The early to mid-1990s were still the era of the retail software stores like Egghead Software and hefty publications like *Computer Shopper*, a large format magazine filled with ads for computer gear and software, that hit over 800 pages at its peak. Consumers were accustomed to buying boxed software, including the aforementioned Windows NT, at prices that easily ran into the hundreds of dollars. For a company accustomed to buying business applications from the likes of Oracle, this might have seemed like a bargain. But not many university students or even

young professionals saw it that way. One of my current colleagues remembers being blown away by the fact that he could buy Red Hat Linux on a CD from a discount retailer for about six dollars. That's cheap.

Eclipsing Unix

Linux was also compatible with Unix programs and skills. A wide range of software development tools were available for it. (Again, for free or cheap.) It had all the applications needed to run it as either part of a big cluster or in a server closet somewhere. The low costs involved also meant that Linux could be and frequently was brought in the back door of companies without IT management even having to know and approve.

By the close of the 1990s, Linux and open source more generally were not yet the dominant force and influence that they are today. But the market share of Linux was already eclipsing Unix on x86 servers. It was running some of the largest supercomputers in the world on the TOP500 list. It was the basis for many of the infrastructure products like “server appliances” sold during the dot-com boom.

And even just by the year 2000, it had attracted thousands of developers from all over the world. The open source development model was working.

Open Source Accelerates

Open source software was born in the 20th century but its great impact has been as a 21st-century phenomenon.

In part, this is because computing changed in a way that was beneficial to open source software such as Linux. Open source has also surely acted as something of a feedback loop to amplify many of those trends.

As we've seen, many of the major early open source projects had an affinity for networked, scale-out computing. Initially, this seemed at odds with the way many enterprise IT departments approached their infrastructure and applications, which tended toward the scale-up and monolithic. Need more capacity? Upgrade your server. Big check, please.

A New Enterprise IT Model

However, by the early 2000s, many organizations were revising how they thought about enterprise IT. As my then-analyst colleague Jonathan Eunice would write in a 2001 research note: “. . . we must understand that what constitutes enterprise computing today is in fact evolving quite rapidly. Every day it moves a little more toward, and morphs a little further into, the realm of network computing. Enterprise IT is increasingly implemented with standardized infrastructure, as well as increasingly is delivered over an Internet Protocol network connection. IT departments increasingly structure themselves, their missions, and their datacenters as do services providers.”

In the world of the Big Iron Unix vendors, an enormous amount of work went into vertical scalability, failover clustering, resource management, and other features to maximize the performance and reliability of single servers. Linux (and Windows) plugged away at features related to these requirements over time. But the world increasingly placed a lower priority on those needs and shifted its attention to the more distributed and network-centric workloads that were closer to the initial open source sweet spot. In fact, there’s an argument to be made that at least some of the Linux development work funded by companies like IBM in the early 2000s focused far too obsessively on making Linux a better scale-up Unix.

Born on the Web

The demand for open source software in the new millennium has also been accelerated by a new class of businesses, which it is no exaggeration to say would not have been possible in the absence of open source. Just one cloud service provider, Amazon Web Services, is reported to have over one million servers. At that scale, presumably Amazon could have licensed the source code for an operating system or other software and then adapted them to their needs. However, like Google, Facebook, and essentially all Internet companies of any size, they primarily run open source software.

In part, this is simply a matter of cost. Especially for those companies offering services that they monetize only indirectly through advertising or other means (think free consumer Gmail from Google), it’s unclear that the economics would work if they needed to pay for software in the traditional historical manner.

That’s not to say that everything these companies use is free. It often makes sense for even technologically sophisticated organizations to establish commercial relationships with some of their suppliers. These companies also require many of the same kinds of

specialized financial and other categories of software and software services all large businesses need. It rarely makes sense to do everything in-house.

Nonetheless, these companies are in a sense almost a new category of system vendor, producing and building much of the software (and even optimized hardware) for their internal use.

Build or Buy?

Still, every company needs to make decisions about where it focuses internal research and development. This was the central thesis of Nick Carr's 2003 article in *Harvard Business Review* entitled "Does IT Matter?" The incredible permeation of software into almost every aspect of business makes some of Carr's specific arguments seem perhaps overstated. (Many felt this was the case at the time as well.) As software plays into competitive differentiation at many levels, it's increasingly rare for companies to treat it as a pure commodity that can all be easily outsourced. However, his broader point that firms should focus on those areas where they can truly differentiate and capture value nonetheless applies.

Open source software makes it easier to trade off build and buy decisions because it's no longer a binary choice. Historically, software tended to be a take-it-or-leave-it proposition. If it didn't work *quite* like you wanted it to, you could put in a feature request to a vendor who might do something about it in a release or two. At least if you were an important enough customer. There was often the option of paying for something custom from your supplier, but that still put a lot of process in the way of getting the changes made.

With open source, companies can choose where to use available open source code unchanged—perhaps with the full support of a vendor. Or they can tweak and augment for their particular needs without the need to build the whole thing from scratch.

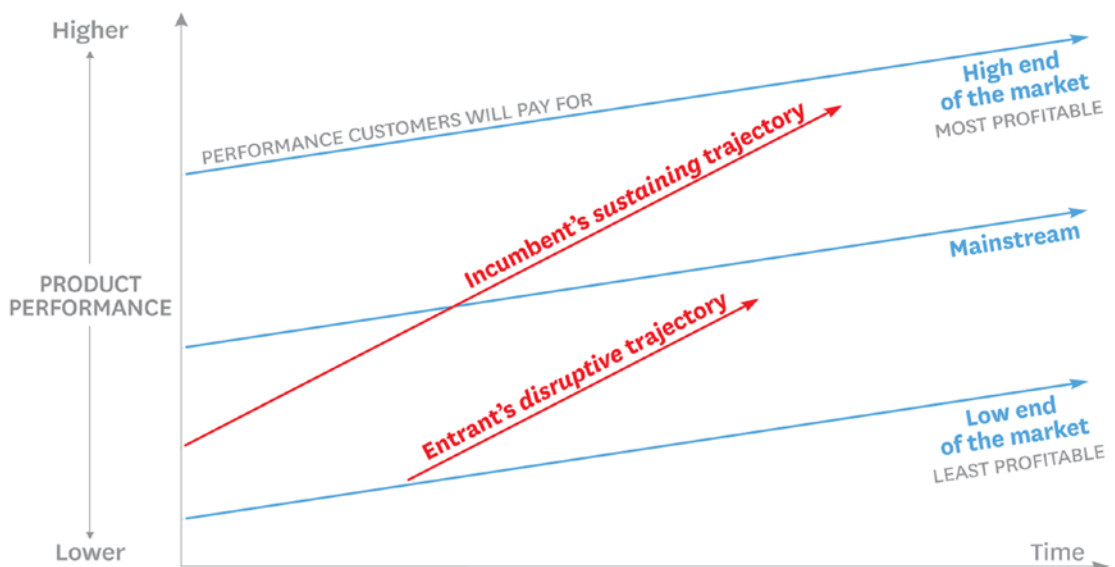
Disrupting the Status Quo

But, really, these arguments about economics, especially in the context of individual companies, skirt around the most important reasons why open source has accelerated so quickly. These reasons involve moving beyond a narrow view of open source as being just about code. It requires thinking about open source as both a very good development model and the means for individuals and companies to work together in ways that just wasn't possible previously.

And that's been the real story of open source in this millenium. It increasingly isn't about being cheaper than proprietary software. Oh, open source software often is. And in some respects, it's classic Disruptive Innovation, which isn't all about price, but usually involves it to some degree (Figure 1-4).

The Disruptive Innovation Model

This diagram contrasts *product performance trajectories* (the red lines showing how products or services improve over time) with *customer demand trajectories* (the blue lines showing customers' willingness to pay for performance). As incumbent companies introduce higher-quality products or services (upper red line) to satisfy the high end of the market (where profitability is highest), they overshoot the needs of low-end customers and many mainstream customers. This leaves an opening for entrants to find footholds in the less-profitable segments that incumbents are neglecting. Entrants on a disruptive trajectory (lower red line) improve the performance of their offerings and move upmarket (where profitability is highest for them, too) and challenge the dominance of the incumbents.



SOURCE CLAYTON M. CHRISTENSEN, MICHAEL RAYNOR, AND RORY MCDONALD
FROM "WHAT IS DISRUPTIVE INNOVATION?" DECEMBER 2015

© HBR.ORG

Figure 1-4. *The Disruptive Innovation model.* Source: Clayton M. Christensen, Michael Raynor, and Rory McDonald, Copyright HBR.ORG.

Disruptive Innovation is the term coined by Harvard Business School Professor Clayton Christensen to “describe a process by which a product or service takes root initially in simple applications at the bottom of a market and then relentlessly moves up market, eventually displacing established competitors.”

We've already seen an example in this book. The personal computer and the servers that evolved from them disrupted the traditional Unix vendors. The Unix vendors themselves often disrupted earlier forms of proprietary systems such as minicomputers.

Christensen also writes that "An innovation that is disruptive allows a whole new population of consumers at the bottom of a market access to a product or service that was historically only accessible to consumers with a lot of money or a lot of skill." Relative to Unix, Linux fits this definition (as does Windows). And, indeed, a great deal of existing Unix business has shifted to Linux over time. This benefited large and technically sophisticated users who simply wanted to spend less money.

But it also allowed new categories of users and businesses that might well not have been able to afford proprietary Unix. Linux has arguably also been a disrupter to Windows by largely keeping it out of network infrastructure and scientific computing markets that it might have eventually captured by default in the absence of an alternative like Linux.

We could apply similar arguments to many other categories of open source software. Databases, application servers and other enterprise middleware, programming tools, virtualization . . . For many examples, the major virtue in their early years was that you could download them off the Internet and try them out for free. They weren't actually better in other respects than the proprietary alternatives.

From Disruption to Where Innovation Happens

However, during the past decade or so, things started to shift. Early cloud computing projects aimed at those who wanted to build their own clouds were probably the first big inflection point. Big data was another. Today, it's cloud-native, containers, artificial intelligence, the many projects related to them, and more. The common thread is that the manner in which open source allows innovations from multiple sources to be recombined and remixed in powerful ways has created a situation in which a huge number of the interesting innovations are now happening *first* in open source.

This is, in part, perhaps because otherwise independent open source communities can integrate and combine in powerful ways.

The Rise of Ecosystems

For example, “big data” platforms increasingly combine a wide range of technologies from Hadoop MapReduce to Apache Spark to distributed storage projects such as Gluster and Ceph. Ceph is also the typical storage backend for the cloud infrastructure project OpenStack—having first been integrated to provide unified object and block storage.

OpenStack networking is also an interesting case because it brings together a number of different communities including Open Daylight (a collaborative software-defined networking project under the Linux Foundation), Open vSwitch (which can be used as a node for Open Daylight), and network function virtualization projects that can then sit on top of Open Daylight—to create software-based firewalls, for example.

The cloud-native space is even more dynamic. It started out being mostly about containers themselves, which are an efficient, lightweight way to isolate applications and application components within an operating system. But it’s expanded with projects like Kubernetes, Prometheus, and hundreds more that encompass container orchestration, monitoring, logging, tracing, testing and building, service discovery, and just about anything else you can imagine might be useful to develop, secure, deploy, and manage distributed applications at scale. We also see the continuing intersection between cloud-native and other areas. For example, both software-defined storage and software-defined networking functionality are being containerized in various ways.

Breaking Up Monoliths

Open source broadly has also evolved in step with a computing landscape that has grown far more distributed and flexible. More integrated too but in a mostly ad hoc and loosely coupled way.

Software both molds the computing environment it runs in and reflects it. A great deal of proprietary software development historically focused on big programs with big license fees that would be (expensively) installed and customized and then left to run for years. Connections between programs and between programs and data required more expensive, complex software going by acronyms like EDI (Electronic Data Interchange). This fit with proprietary business models.

Complexity was actually a good thing (at least from the vendor’s perspective). It tied customers to a single vendor, generated consulting fees, and created lots of upsell opportunities with all the options needed to make everything work together. Complexity also meant that there was no real way to determine upfront if software was going to

work as advertised. In practice, I've seen more than one high-end computer systems management program end up as a very expensive bookend sitting on a shelf because it never really quite did the job it was purchased to do.

Even once the Web came onto the scene, initial efforts to have less rigid integrations still reflected traditional ways of doing things. For example, Service-oriented architecture (SOA), at least in its initial form. Many of the core ideas behind SOA, such as separating functions into distinct units that communicate with each other by passing data in a well-defined, shared format, are part of modern architectural patterns such as microservices. However, as implemented at first, SOA was often mired down by heavyweight web services standards and protocols.

By contrast, today's distributed systems world is more commonly characterized by lightweight protocols like REST; open source software components that can be mixed, matched, and adapted; and a philosophy that tends to favor coding reference implementations rather than going through heavyweight standards-setting processes. Open source has brought Unix-flavored approaches like highly modular design to both platform software and applications more broadly.

Linux and Open Source Had Arrived

In 2003, IBM aired a TV commercial titled "Prodigy." It featured a young boy sitting and absorbing pearls of wisdom. Hollywood director Penny Marshall: "Everything's about timing, kid." Harvard professor Henry Louis Gates: "Sharing data is the first step toward community." Muhammad Ali: "Speak your mind. Don't back down." The commercial ends with someone asking who the kid is. "His name is Linux."

The ad, created by Ogilvy & Mather, was a don't-change-the-channel ad with arresting imagery. Then head of IBM's worldwide advertising, Lisa Baird, said it was targeted at "CEOs, CFOs, and prime ministers." The investment in this ad reflected how forward-looking individuals and organizations were starting to view open source.

Irving Wladawsky-Berger, who recognized the potential of Linux early on and ran Linux strategy for IBM at the critical turn-of-the-century juncture, noted in a 2011 LinuxCon keynote that "We did not look at Linux as just another operating system any more than we looked at the Internet as just another network." He went on to say that we viewed it as "a platform for innovation into the future, just like we viewed the Internet."

Equally notable were the smaller companies like Red Hat, SUSE, and others who were starting to build businesses that not only used but were explicitly based on open source.

It may not have been clear to many in 2000 that Linux was anything more than a Unix-like operating system with an unusual license. Or that open source more broadly was a significant part of the software or business equation. However, here in 2018, it's clear that Linux and open source broadly are playing a major role in pushing software innovation forward. And that really means pushing business capabilities forward given how inextricably linked they are to software.

CHAPTER 2

From “Free” to “Open Source”

We’ve seen how free and open source software not only came onto the scene but became an essential part of the computing landscape. Not just cheaper than proprietary software but a model that encourages the creation of software ecosystems and new innovations.

However, in the process of telling this story, I’ve moved quickly past some details. Are the oft-conflated “free” and “open source” one and the same, or do they capture an important distinction? What do you need to know about the legal basis for this type of software without getting too deeply into law, licenses, and contracts? Is it safe? How does this open source thing work if you want to participate?

This chapter takes you from the context to today’s on-the-ground reality.

Words Can Matter

Richard Stallman and his Free Software Foundation have long emphasized the word “free.” But Stallman didn’t intend that to mean someone had to give you a copy of software without charging for it. Rather, as he would later add in a footnote to the GNU Manifesto: “I have learned to distinguish carefully between ‘free’ in the sense of freedom and ‘free’ in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies—and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.”

The oft-repeated shorthand is that free software is about free as in freedom rather than free as in beer.

Why Free

The GNU Manifesto was a reaction to what was happening in the computer industry more broadly at the time. Convivial academic collaborations were giving way to fragmented and proprietary commercial products.

As Steven Weber writes in *The Success of Open Source* (Harvard University Press, 2005), “The Free Software Foundation was fighting a battle with this narrative on philosophical grounds. To reverse the decline, Stallman sought to recapture what he thought was essential about software—the ideology of cooperative social relations embedded within the knowledge product. The philosophy implanted in the software was as important as the code.”

There were a variety of terms floating around for this new type of software. Brian Behlendorf of the Apache web server project, an early open source software success that became an important element in the Internet buildout, favored “source-code available software.” But free software became the common term. And, for newcomers, distinguishing free as in freedom from free as in beer was often confusing.

The Coining of “Open Source”

In 2018, writing on the 20th anniversary of the coining of the term “open source,” Christine Peterson—who had been the executive director of the Foresight Institute—recounts how she was focused “on the need for a better name and came up with the term ‘open source software.’ While not ideal, it struck me as good enough. I ran it by at least four others: Eric Drexler, Mark Miller, and Todd Anderson liked it, while a friend in marketing and public relations felt the term ‘open’ had been overused and abused and believed we could do better. He was right in theory; however, I didn’t have a better idea, so I thought I would try to go ahead and introduce it.”¹

At a meeting later that week, the terminology again came up when discussing promotion strategy related to Netscape’s plan to release its browser code under a free software type of license. A loose, informal consensus developed around the open source term.

Influential publisher and event organizer Tim O’Reilly would soon popularize it. In addition to using the term himself, “on April 7, 1998, O’Reilly held a meeting of key leaders in the field. Announced in advance as the first Freeware Summit, by April 14 it was referred to as the first ‘Open Source Summit.’”

¹<https://opensource.com/article/18/2/coining-term-open-source-software>

Pragmatism and Commercialism

The initial motivation for this shift in terminology was primarily practical. People were tired of explaining that it was perfectly well and good to charge for “free software.”

However, it served another purpose that one suspects many proponents such as O’Reilly realized from the beginning. It helped to distance open source as a broader movement for organization, collaboration, innovation, and development from the narrower and more philosophical bent of free software.

O’Reilly would note the following year that “the Free Software Foundation represents only one of the traditions that make up the open source movement” with university software development traditions—most notably Berkeley’s BSD—also being significant. Furthermore, much focus around free software ended up being around licenses, which may have been important (especially at first) but were only one component of broader discussions around openness and user freedoms.

The shift in terminology also reflected how open source software, including but hardly limited to Linux, was becoming commercially interesting to existing vendors, new vendors, and end users. Some traditional IT suppliers like IBM were making huge investments in open source. New companies that were inextricably linked to open source were aborning.

The first Internet wave in the latter half of the 1990s was built on open source. Large-scale hosting providers, those building specialized software appliances for serving email and web pages, and the early iterations of search engines and other now familiar services all required open source software to function. Many wanted to customize the software they depended on. Furthermore, the scale at which they operated increasingly made alternatives to proprietary software an economic necessity.

Commercialism led to something of a schism between hobbyists and those focused more on the ideological roots of free software on the one hand and the more pragmatic and profit minded on the other. A focus on free as in freedom software versus software developed collaboratively and in the open because it was more effective to do so. The two aspects were by no means mutually exclusive, but there was clearly tension for both philosophical and pragmatic reasons.

Today, one interpretation is that the two perspectives have mostly reached a rapprochement; the practical benefits of open source software include user flexibility and freedoms that at least overlap their ideological counterparts. A more cynical view is that pragmatism and profit have come to dominate open source—at least with respect to

where much of the attention and effort flows. In reality, it’s a bit of both and whether it’s closer to one or the other is going to depend on your priorities and point of view.

How Open Source Licensing Works

The choice of licenses is usually a less contentious topic than it once was. But licenses as a legal concept remain an important part of open source software’s foundations. As a result, it’s useful to broadly understand their role and some of the key distinctions among them that can affect how open source development works.

While it doesn’t control the use of the “open source” term, the open source definition and set of approved licenses maintained by the Open Source Initiative (OSI) serves as a generally accepted standard for what constitutes an open source software license. Core principles include free redistribution, providing a means to obtain source code, allowing modifications, and lack of prohibitions about who can use the software and for what purpose.

The last point means that you can’t say, “This software is free to use for educational purposes only.” Open source has succeeded in part because approved open source licenses don’t place restrictions on how you can use software you’ve obtained from some source. Depending upon the license, you may have obligations if you, in turn, redistribute the software or incorporate into some else that you then distribute. But use is fair game.

Do You Have to Give Back or Not?

There are two broad categories of licenses. One includes “copyleft” or reciprocal licenses, of which the General Public License (GPL) is the best known and is the one used by Linux. The other includes “permissive” licenses, most notably the Apache, BSD, and MIT licenses.

There are a variety of other licenses as well. Some are essentially legacy licenses that have just never been retired. Others are designed to be more suitable for other copyrighted material such as books or photographs. For example, the CC BY-SA Creative Commons license variant requires attribution but does not prohibit use or remixing.²

²Other Creative Commons license variants impose restrictions such as NC (noncommercial use only), which do not conform to commonly accepted open source software license standards.

It’s worth emphasizing that material released under an open source license is still copyrighted. Indeed, copyright law is integral to the working of open source licensing that grants users various rights that they wouldn’t have under a default “All Rights Reserved” copyright. In most countries, creative works—including computer software—are automatically copyrighted as soon as they’re created.

There are some exceptions. For example, US copyright law places works of the US federal government in the public domain upon creation. An individual can also choose to place their own work in the public domain, although this is controversial for rather arcane legal reasons. The Creative Commons website released a copyright waiver in 2009 called CC0 as an alternative to a public domain for this reason.

Different licenses impose more, fewer, or different types of restrictions within that general framework. But copyleft and permissive capture the core distinction.

Protecting the Commons

A copyleft license requires that if changes are made to a program’s code, and the changed program is distributed outside an organization, the source code containing the changes must likewise be distributed (or otherwise made reasonably available). Permissive licenses don’t include that requirement.

Copyleft says that you can’t take someone’s code, change it or mix it up with other code, and then ship the resulting program in machine-readable form without also making it available in human-readable source code form. There’s a philosophical point here. If you take from the commons—that is, use open source software someone else has created—you also have to give back to the commons if you distribute it.

This reciprocity requirement had roots in the practical. In a software world that was seemingly becoming increasingly proprietary and profit seeking, the thinking went, why *wouldn’t* corporations naturally vacuum up code from the commons, give little in exchange, and effectively become free-riders.

After all, the “tragedy of the commons” was a social science phenomenon articulated way back in 1833 by the British economist William Forster Lloyd, from the hypothetical example of unregulated grazing on common land in the British Isles. Better to require reciprocal contributions, especially given that the advantages of open source as a development model mostly hadn’t yet been articulated and proven.

But it was also just a reflection of a software movement that was as much about philosophical principles as it was practical results.

A couple of things that cause confusion are worth highlighting. Let’s look at them now.

Seeing Through the Copyleft Mire

The first is that precisely defining “mix it up” gets into technical and legal details that, absent substantial case law, are a matter of some debate. For example, some argue that the manner in which two programs are combined together—that is, whether they’re dynamically or statically linked in Unix parlance—makes a difference. Others say it doesn’t.

However, to the degree that there’s a risk, it’s usually not so much that it’s hard to determine whether code is actually being mixed because it runs into an ambiguous edge case. Rather, it’s that code under a copyleft license was deliberately but carelessly used in projects where it wasn’t appropriate or was against company policies.

The other point is that copyleft licenses are specifically about *distribution*. Again, there are some nuances but, essentially, if you distribute software, for profit or otherwise, by itself or as part of a hardware product, you must make the source code for the work that incorporates the copyleft code available. In other words, so long as software is used internally, there’s no requirement to distribute the code.

Permissive Licenses Gain

We’re seeing an ongoing shift to more permissive licenses. Matthew Aslett of market researcher 451 Group wrote in 2011 that “2010 was the first year in which there were more companies formed around projects with non-copyleft licenses than with strong copyleft licenses.”

More recent data shows a continuing trend, as do the anecdotal observations of industry observers.

Black Duck by Synopsys, which automates security and open source license compliance, maintains a Knowledge Base of over two million open source projects. As of 2018, among projects using the top five licenses accounting for 77 percent of the total projects, about two-thirds used a permissive license (MIT, Apache 2.0, or BSD).

There are some differences among these permissive licenses, primarily with respect to the types of copyright and other notices that must be retained and displayed. Patent language, both explicit and implied, also differs; newer licenses are more likely to call out patents specifically in the text of the license. These differences will be important to many organizations shipping products that make use of open source code. However, we can mostly think of these permissive licenses as giving permission to use code covered by such licenses without meaningful restriction.

In general, this shift reflects a lessening concern about preventing free-riders and more on an increasing focus on growing communities.

There are indeed many free-riders. That’s a given. But open source has been widely embraced by all manner of companies because they’ve found that open source is a great way to engage with developer and user communities—and even with competitors. It’s emerged as a great model for developing software and capturing innovation wherever it’s happening.

Furthermore, in a cloud services world, the GPL doesn’t even protect against free-riding especially well. If you sell me a service delivered through a web page rather than software I download, that’s not distribution from the perspective of the GPL. Yet, this is the increasingly dominant way through which you use many types of software. [Salesforce.com](https://www.salesforce.com). Amazon Web Services. Microsoft Azure. Google Cloud Platform. All you need is a web browser to use any of them. You don’t need software that is distributed in the traditional sense of shipping bits on disk or making them available for download.

To some this is indeed a loophole and the Affero General Public License, which redefines the meaning of distribution to include software delivered as a service, was introduced to close it. But it’s not widely used.

Driving Participation Is the Key

What’s usually more important is decreasing the barriers to participating in projects and collaborating across companies. And while individuals and organizations participate and collaborate on projects licensed under the GPL, most famously Linux, permissive licenses are increasingly viewed as an often-better choice.

In part, this is because companies developing with a combination of open source and closed source code simply want to maintain the flexibility to decide what code to contribute and when based on their own desires and not the demands of a license. It also reduces license incompatibility issues. Software licensed under major permissive licenses can be added to GPL-licensed code, which can then be distributed under the GPL, but the reverse doesn’t apply because the GPL is more restrictive than a license like MIT. (Combining code can’t result in removing obligations such as reciprocity.)

Whatever the reason in an individual case, it’s about maximizing participation in projects. The Eclipse Foundation’s Ian Skerrett argues that “projects use a permissive license to get as many users and adopters, to encourage potential contributions. They aren’t worried about trying to force anyone. You can’t force anyone to contribute to your project; you can only limit your community through a restrictive license.”

Appropriate licensing remains relevant in a table stakes sort of way for open source projects. However, it’s no longer a major focus for creating successful communities, projects, and businesses. As Chris Aniszczyk of the Cloud Native Computing Foundation puts it: “Licensing and all that is table stakes. That’s a requirement to get the gears going for collaboration, but there are [other] aspects around values, coordination, governance.”

Maintaining Open Source Compliance

Maintaining compliance with these different types of open source licenses can sometimes seem intimidating, but it’s mostly about having established processes and following them with reasonable care.

Putting Controls in Place

The Linux Foundation recommends having a designated open source compliance team that’s tasked with ensuring open source compliance.³ Such a team would be responsible for open source compliance strategy and processes to determine how a company will implement these rules. The strategy establishes what must be done to ensure compliance and offers a governing set of principles for how employees interact with open source software. It includes a formal process for the approval, acquisition, and use of open source; and a method for releasing software that contains open source or that’s licensed under an open source license.

As open source becomes more widely used within many companies, it’s increasingly important to have these kinds of controls in place anyway for reasons that aren’t directly related to license compliance. Unvetted code from public repositories may not be current with security patches or may otherwise not meet a company’s standards for production code.

What Are Your Policies?

A first step is to establish a process and appropriate policies. Often this includes a list of acceptable licenses for software components used for different purposes and in different roles. For example, companies widely use commercial enterprise software that

³<http://www.linuxfoundation.org/using-open-source-code/>. Excerpts under Creative Commons Attribution ShareAlike 4.0 International License.

includes programs licensed under the GPL like Linux. However, they may choose to only incorporate open source software components that use permissive licenses into their own software. Or they may be fine with GPL components for their internal software but not in products that they ship or otherwise expose to partners and customers.

In any case, these are matters of policy for management, including the legal staff. One important general recommendation, however, is to not make things too complicated. As Ibrahim Haddad, vice president of R&D and head of the Open Source Group at Samsung Research America, puts it: “If your code review process is overly burdensome, you’ll slow innovation or provide a good excuse for developers to circumvent the process completely.”

An Ongoing Process

The ongoing process then uses scanning tools—whether proprietary or open source—to determine the software components in use, the licenses of those components, potential license conflicts, and any dependencies. Related tools can be used to identify known security vulnerabilities. Problems can then be identified and resolved. For example, a proprietary software component linking to a GPL-licensed component might be in compliance with policy for an internal tool but should raise a flag if it’s going to be shipped externally.

Projects versus Products

To this point, I’ve been mostly talking about open source software as a singular collection of bits. For a hobbyist project, that may be a reasonable simplification. For Linus Torvalds, it was bits on a desktop computer at the University of Helsinki. These days, it’s more commonly files stored on an online repository like GitHub. No one is selling the software. No one is promising support in any sort of formal way.

However, for most products that companies sell, it’s important to distinguish between projects and products.

Upstream and Downstream

In its simplest form, there is an “upstream” community project and a “downstream” product based on the upstream. For this discussion, assume that the product is also fully open source software although some companies practice partial open source development by combining open source components with proprietary ones; “open core”

is a common form of this practice. Downstream products may also bring together and integrate independent and quasi-independent upstreams but the same general principle applies.

Upstream is the catch-all term for the ultimate source of the project. It's the core group of contributors, their mailing lists, website, and so on. Ideally, it's where most of the community development happens. By contrast, the product is something that a customer buys to solve a business problem.

Innovation, rate of improvement, wide acceptance, and other aspects of the product may derive in large part from the fact that a vibrant upstream community exists. But don't confuse the two. As Paul Cormier, Red Hat's president of Products and Technologies, puts it: "Too often, we see open source companies who don't understand the difference between projects and products. In fact, many go out of their way to conflate the two."⁴

Usually, the open source project comes first. Sometimes the process gets reversed when a company acquires a proprietary product and releases it as open source. But most of the same principles apply to the relationship of the project to the product as seen in Figure 2-1.

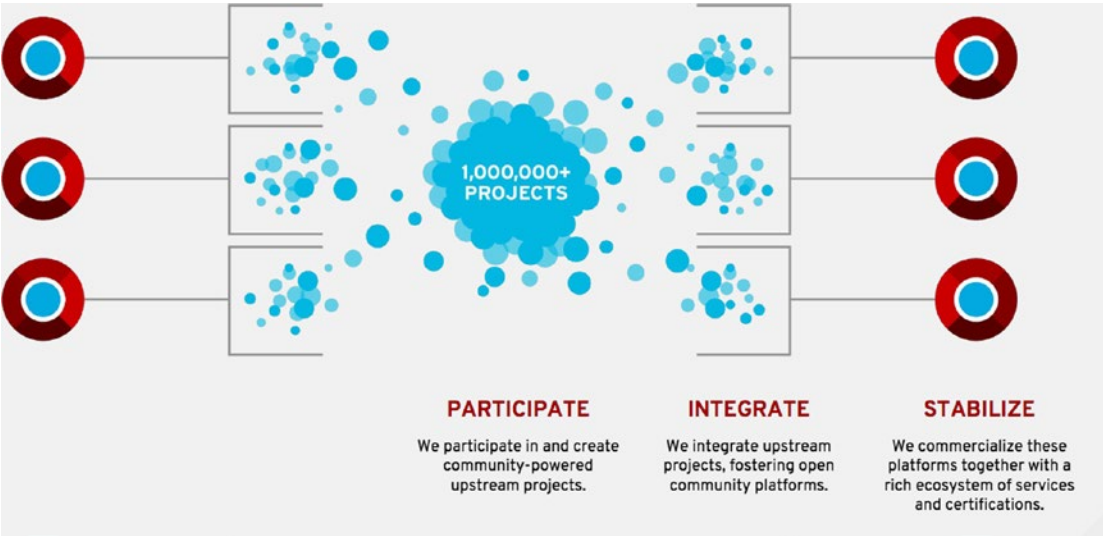


Figure 2-1. The open source product development process. Source: Red Hat.

⁴<https://www.redhat.com/en/blog/what-makes-us-red-hat>

Projects and Products Depend on Each Other

To be clear, there are strong linkages between successful projects and successful products.

For example, as I’ll cover in more detail later, metrics for community success often include measures of community breadth; how many contributors work for someone other than the project’s creator? The reason this is important is that a community with no outside contributors is effectively an open source project in name only. Others can look at the code, but there’s none of the broader participation that makes the open source development model work so well.

It’s also good to have a strong relationship between project and product from a development perspective. “Upstream first” is a best practice, meaning that code changes preferably go into project from whence they subsequently flow downstream into the product.

It’s a best practice, in part, because it means less work. There’s less code to maintain that’s not part of the upstream. This isn’t always possible. The most vibrant and independent projects won’t always accept code that, for example, is specific to a particular customer’s product requirement. But companies that work effectively with and participate in upstream projects stand to have the most influence when it comes to decisions that are important to them.

What Support Means

However, many attributes are part of a product that aren’t necessarily in a project. Many assume that the main difference is that people pay for support in the case of a product. That’s not wrong; customers still call and email for support in the case of software products based on open source just as they do for software products that aren’t based on open source. But, to use Red Hat enterprise software subscriptions as an example, the product includes much more, including some things that are largely specific to open source software.

Support itself has broadened since the days when it meant picking up the phone to get a question answered.

When things go wrong in a production software environment, the ability to access the right information quickly can be the difference between a fast return to normal operations and a costly outage. And sometimes, the best support call is one you don’t need to make. For example, customers today like to search to locate articles, technical

briefs, and product documentation that are most relevant to the problem at hand. System architects like to browse detailed technical case studies that engineers have designed, tested, and benchmarked.

Reducing Risk

A big part of what enterprise software companies are looking for is reduced risk.

In addition to the support itself, this falls into several different areas.

The first is life cycle support. In a project it's typical to have an unstable branch and a stable branch. At some point of development, the unstable branch is deemed to be stable, it replaces the current stable branch, and the cycle begins again. At that point, the prior stable branch is typically retired. It's still available, but typically work on it is frozen. (This isn't a hard and fast rule; sometimes developers will retrofit a fix for a particularly serious bug but, generally speaking, community projects tend to focus on the latest and greatest.)

However, enterprise customers often want long life cycles. This can mean five years, seven years, or even longer. Longer life cycles mean more choice and flexibility, reduced cost and risk, and more ease of planning. To meet this requirement, enterprise software companies “backport” fixes and feature enhancements into older versions of their software.

Certifications are also an important part of an enterprise software product. This includes certifying hardware, certifying software, and certifying providers like public clouds. These types of certifications are based on joint testing with partners, established business relationships, and other agreements to both reduce the number of potential issues and to have processes in place to resolve problems when they happen.

While well-run projects incorporate automated tests and other processes intended to reduce the number of bugs introduced into the code base, downstream products tend to have more robust quality assurance testing. For example, Red Hat's program includes acceptance, functionality, regression, integration, and performance testing.

Other product features can include legal protections such as defending customers against intellectual property lawsuits or dealing with certain other legal issues.

The Intersection of Security and Risk

A final area of product assurance that is often top of mind today is security. Some aspects of open source product security are similar to those associated with enterprise software products generally. It's increasingly important to have a dedicated team of engineers

who proactively monitor, identify, and address potential risks. This lets them give accurate advice to quickly assess risk and minimize business income.

What’s different with open source is that developing software in collaboration with users from a range of industries, including government and financial services, provides valuable feedback that guides security-related discussions and product feature implementations. And, in the case of security vulnerabilities with broad impact, it allows engineers from multiple companies to develop fixes cooperatively.

Before leaving the topic of security though, it’s worth considering it more broadly. The intersection of open source and security still causes a lot of angst and misunderstanding.

Securing Open Source

This discussion requires teasing apart a couple of different concepts.

The first gets back to projects versus products. This is pretty straightforward even if, as noted earlier, it is still often a source of confusion.

Business as Usual: Patches and Advice

An open source software vendor needs to provide security patches and advice in much the same manner as any other software vendor—though the open source company has the added advantage of being able to collaborate closely with customers, partners, and other vendors.

Many projects also do an effective job of developing and making available security fixes and best practices—especially when fixes are pushed upstream quickly—but the process may take longer and be less formalized.

The second is open source security in the abstract. In other words, does access to source code by both the “good guys” and the “bad guys” help or hurt security? When people ask whether open source is more or less secure, this is what they are often talking about.

One can get a sense of the debate from two different statements attributed to the US Department of Homeland Security’s Luke McCormack in 2016. A week after his statement that opening up federal source code would be like giving the Mafia a “copy of all FBI system code” set off a minor firestorm, he walked it back to say, “Security through obscurity is not true security: we cannot depend on vulnerabilities not being exploited just because they have not been discovered yet.”

Does Code Help the Bad Guys?

The hurts-security side of the argument is rooted in physical analogies.

When many people think about security, they probably think about something like a home security system. Physical security systems do typically depend to at least some degree on “security through obscurity.” They may actively protect against a wide range of threats, but they probably also depend on, to at least some degree, the would-be burglar not knowing precisely what types of sensors there are, where they are placed, and how the property is being monitored.

The degree to which obscurity makes software more secure is controversial, and the answer probably boils down to “it depends” to some degree. As Daniel Miessler has noted, “It’s utterly sacrilegious to base the security of a cryptographic system on the secrecy of the algorithm.”⁵ At the same time, he argues that “The key determination for whether obscurity is good or bad reduces to whether it’s being used a layer on top of good security, or as a replacement for it. The former is good. The latter is bad.”

Even the US National Institute of Standards and Technology (NIST) equivocates. On the one hand, it has stated that “System security should not depend on the secrecy of the implementation or its components.” However, they also recommend (in the same document) that, “For external-facing servers, reconfigure service banners not to report the server and OS type and version, if possible. (This deters novice attackers and some forms of malware, but it will not deter more skilled attackers from identifying the server and OS type.)”⁶

The general consensus seems to be that obscurity doesn’t help much (if any) and you shouldn’t depend on it in any case. Attacks mostly come through probing for weaknesses en masse. They exploit configuration errors, default passwords, and *known* vulnerabilities that haven’t been patched yet. Of course, any vulnerability that a white hat security researcher finds by combing through source code could be found by a black hat one first. But that’s not anything like a common pattern.

Or Is “Many Eyes” the Secret Sauce?

At the same time, it’s not clear to what degree the common argument favoring open source for security reasons—“with many eyes, all bugs are shallow”—applies either. One

⁵<https://danielmiessler.com/study/security-by-obscurity/>

⁶<http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf>

problem is that many eyes can still miss things. The other is that not all projects have many eyes on them.

In early 2017, I sat down with then-CTO of the Linux Foundation Nicko Van Someren to talk about the Core Infrastructure Initiative (CII), a group set up in the wake of the Heartbleed bug, a security vulnerability that potentially affected of about 70 percent of the world’s web servers.

In the case of Heartbleed specifically, the bug (discovered by a Google engineer) was quickly fixed, but it exposed the fact that a number of key infrastructure projects were underfunded. As Van Someren put it: “Probably trillions of dollars of business were being done in 2014 on OpenSSL [the software with the Heartbleed bug], and yet in 2013, they received 3,000 bucks worth of donations from industry to support the development of the project. This is quite common for the projects that are under the hood, not the glossy projects that everybody sees.”

He went on to say that “We try to support those projects with things like doing security audits where appropriate, by occasionally putting engineers directly on coding, often putting resources in architecture and security process to try to help them help themselves by giving them the tools they need to improve security outcomes. We're funding the development of new security testing tools. We're providing tools to help projects assess themselves against well-understood security practices that'll help give better outcomes. Then, when they don't meet all the criteria, help them achieve those criteria so that they can get better security outcomes.”

Thinking Differently About Risk

The best that we can probably say is that software being open source is neither a hazard nor a particular panacea. And it’s probably not a very useful debate.

Patrick Heim, CISO, ClearSky Security argued at the 2018 Open Source Leadership Summit that “Maybe [we] need to move beyond the argument of which is better. How do we live in this new world where there is more open source? We have to think slightly differently about how we manage risk.” Many organizations provide frameworks for thinking about cybersecurity. Figure 2-2 provides an example from NIST.

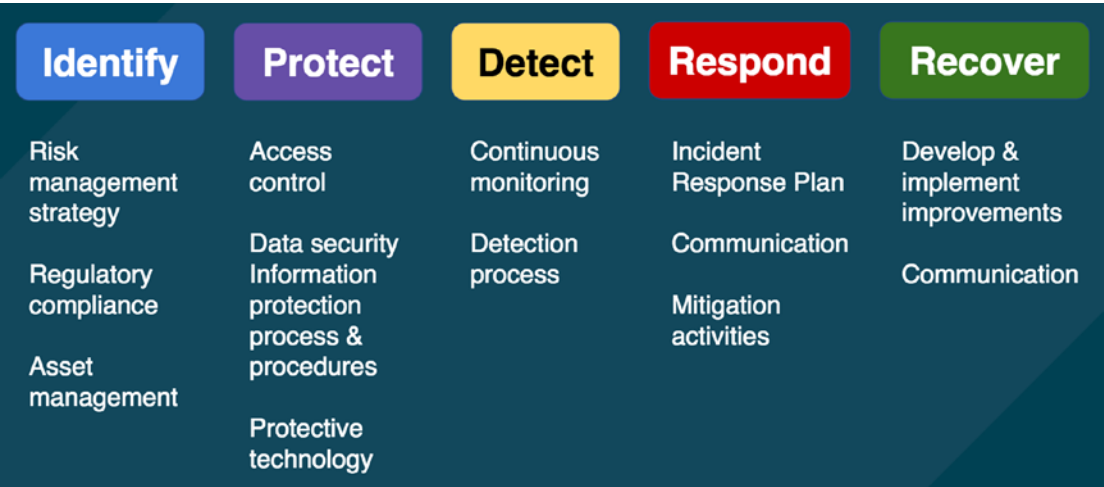


Figure 2-2. The *National Institute of Standards and Technology* (NIST) cybersecurity framework focuses on using business drivers to guide cybersecurity activities and considering cybersecurity risks as part of the organization’s risk management processes.

This includes a lot more automation. A lot more monitoring. A lot more understanding of a software supply chain that increasingly will include open source components. The license considerations I mentioned earlier are only a small part of this. Equally, really more, important are questions such as whether a software component is being maintained, who is maintaining it at what level of activity, who is vetting it for trustworthiness?

And, ultimately, it’s about making risk-based decisions. That means having an honest conversation with the business about priorities and risk management.

Participating in Open Source Projects

Before moving onto how the open source software development process works in more detail—how communities work, how they’re governed, what the process looks like—let’s consider what getting involved in open source looks like.

We can approach the question of participation from a number of different angles. Here are three.

- Starting an open source project associated with a current or planned software product;
- Doubling down an existing open source project;

- Creating an open source program office to manage internal open source use and external contributions—including creating new projects.

There are common themes to these. One of those is that participating in open source is not a philanthropic endeavor or certainly doesn’t need to be. Rather, as the Linux Foundation’s Jim Zemlin puts it:

The epiphany that many companies have had over the last three to four years, in particular, has been, “Wow. If I have processes where I can bring code in, modify it for my purposes, and then, most importantly, share those changes back, those changes will be maintained over time. When I build my next project or a product, I should say, that project will be in line with, in a much more effective way, the products that I’m building. To get the value, it’s not just consumed, it is to share back and that there’s not some moral obligation (although I would argue that that’s also important), there’s an actual incredibly large business benefit to that as well.” The industry has gotten that, and that’s a big change.

Another theme is that there’s no template. There are principles and practices that recur over and over in most successful open source projects. Many of which should be regarded as more than guidelines but less than an absolute set of rules. However, while many open source projects fall into certain recurring patterns, there’s no template. Chris Aniszczyk calls it the Tolstoy principal for open source, “Each project is unhappy in its unique own way.” (By “unhappy,” read has singular concerns, needs, and challenges.)

Starting an Open Source Project

The lines between the developers of software and the consumers of software are increasingly blurred. So is the demarcation between those who make money from software and those who make money from doing other things like building widgets; companies increasingly deliver software services and experiences as well as physical things.

But it’s still useful to treat open source projects that are directly tied, now or in the future, to commercial software products as distinct from projects created by organizations with other purposes in mind.

One of the first questions to ask, though, is one of the most important. Do you need to start a new project? In some cases, yes. Many commercial software products start life as a new open source project. In other cases, there’s an existing commercial software product and the company or an acquirer of the company wants to start a project based on the product’s code. At Red Hat, for example, when we acquire companies and products, we routinely go through a process to make all components open source. In still other cases, a software company has a new engineering initiative that it wants to develop in open source, but there’s no existing project that’s a natural fit.

Some of the most successful open source projects have come about because companies and individuals decided to rally around an existing project rather than going off and doing their own thing. This sort of behavior is, after all, central to the open source development model. But let’s assume that you have good reasons to create a new project. What are some of the things you’ll want to think about next?

What success looks like for the community project is a good place to start. This should correlate with the reasons you have for creating a new project. What are you trying to accomplish? What are your business objectives?

Typically, the project should support the success of a product offering based on the project or otherwise support commercial objectives such as creating brand association for the sponsoring company. If, on the other hand, it’s a pure community project that’s not associated with a commercial product, it should be able to stand on its own as a viable open source community project.

From there, you can move on to considering project-specific needs for a community launch. These can include existing closed source code that you plan to open source; source code licensing; and how the project will be positioned relative to downstream products from marketing, trademark, naming, and other perspectives.

Will the project be a stand-alone project supported by a company or a group of companies or does it make more sense to become part of an existing foundation? Dan Kohn of the Cloud Native Computing Foundation (under the Linux Foundation) took a look at the 30 “highest velocity” open source projects—as measured by commits, pull requests, and issues.⁷

Nine were backed by foundations including Linux (Linux Foundation), three OpenStack projects (OpenStack Foundation), and Kubernetes (CNCF). Fourteen were backed by companies including Google, Facebook, Red Hat, Elastic, and Basecamp. Six

⁷<https://www.linuxfoundation.org/blog/successful-open-source-projects-common/>

projects were not mainly backed by one company or software foundation; their nature varied but a number were characterized by their value mostly coming from “recipes” individually contributed and updated by hundreds of independent contributors.

In other words, it’s quite a mix.

As with many aspects of open source communities and projects, the “right” approach is very situational. Foundations—whether created for a specific project or with broader scope—are often seen as a more neutral collaboration point than projects solely under the control of a single company. However, as seen above, they’re not a prerequisite for successful projects. It just often takes more work to attract and retain outside contributors and raises the bar for demonstrating that processes for communicating, making decisions, and accepting contributions aren’t biased in favor of the main project sponsor.

In addition to where the project will live, there’s the question of how it will be governed. Some foundations, such as the Apache Software Foundation, have a fairly standard structure for projects. (At least ostensibly. The on-the-ground realities aren’t always so neat and clean.) Others, like the Cloud Native Computing Foundation, take a more flexible approach based on the considerations of individual projects. Still others, like the OpenStack Foundation, were explicitly started to meet the needs of a specific project and its related subprojects.

At a detailed level, there are many additional decisions. Where will the code be hosted? How will communications take place? How will a selected governance model be implemented at a detailed level? Who makes the ultimate decisions and who gets to commit code? How will you enable people to move from user to contributor? Is everything lumped together or will you create Special Interest Groups (SIGs)? Will you have a dedicated community manager? What’s your time line?

And, of course, what’s the project’s name and what does the logo look like? (Topics that always seem to consume an outsized portion of time.)

Finally, iterate, iterate, iterate.

Doubling Down an Existing Open Source Project

However, organizations can often most effectively participate in open source development by joining an already existing community. Doing so can come with its own set of challenges if the community is dominated by companies or individuals with interests that run counter to your own. Nonetheless, connecting to an existing project is often easier, quicker, and more certain than trying to kickstart a new initiative.

Participation in open source communities doesn’t need to be about coding. For example, companies can help with testing in their specific environments, which will often be at higher scale points or otherwise different from configurations that can easily be tested in a lab. This sort of participation in software development by end users was commonplace even before open source software became so widespread. We typically referred to it as alpha or beta testing depending upon how early in the process you were involved.

Writing documentation and providing funding are also ways to get involved.

However, as longtime IBM Software Group head Steve Mills put it: “Code talks.” The Linux Foundation also argues that the greatest influence in open source projects is through the quality, quantity, and consistency of code contributions.⁸

Stormy Peters, senior manager for Community Leads at Red Hat observes that “For many larger projects, we know that most of our contributors are going to be people who work at companies that need to use projects like Ceph and Gluster. We have customers, and customers often contribute to software because they’re using it. We consider both the individual participation and the company participation as success stories.”

Some companies may participate in open source software development as a way to give back to the community. However, there are plenty of business justifications as well.

A big reason is to take advantage of the open source development process. While it’s often possible to use and modify open source code without making contributions, “forking” a project in this way “defeats the whole purpose in terms of collective value” in Jim Zemlin’s words. He adds that “You’re now, basically, supporting your own proprietary fork. Whether or not it’s open source, it doesn’t matter at that point. No one understands it but you.” Microsoft’s Stephen Walli describes it as being “very expensive to live on a fork.”

Stormy Peters and Nithya Ruff, senior director of Open Source Practice at Comcast, add that “Companies that start fixing bugs or adding new features and functionality to an open source project without contributing them back into the upstream project quickly learn that upgrading later to add new features or apply security fixes can be a nightmare that drives maintenance costs through the roof. Contributing your changes back into the upstream project means that they will automatically be included in future updates without incurring additional maintenance costs.”⁹

⁸<http://www.linuxfoundation.org/resources/open-source-guides/improving-your-open-source-development-impact/>

⁹<https://www.linuxfoundation.org/participating-open-source-communities/>

Furthermore, coming back to Mills’s talking code, new features and functionality come from code contributions, and those contributions can influence the direction of the project. If you want the project to have specific functionality that you need, it may be up to you to implement potential changes.

That said, working with open source projects requires awareness of and sensitivity to the norms, expectations, and processes associated with the specific community. While your contributions, of whatever form, will often be welcomed, that doesn’t mean just showing up and throwing weight around. You need to be aligned with both the overall direction of the project and the way that the project and its associated community operates. A good approach is to have someone join the community and spend some time observing. Alternatively, you can hire someone who already has a proven track record of participation in the community.

Communities have different characters, ways of participating, and different channels for communication—which include mailing lists, forums, IRC, Slack, bug trackers, source code repositories, and more. These are useful for both ongoing communications and to understand how a community works before jumping in. “Lurk first” is what Peters and Ruff advise because “the more time you spend reading and listening, the more likely it is that your first contribution will be well received.”

They also counsel reading up on the project governance and leadership before contributing. Who makes the decisions for various types of contributions? Is there a contributor licensing agreement? What’s the process for contributing? What classes of contributors are there? I’ll get into the specifics of different governance and community models in the next chapter. But, at a high level, it’s important to appreciate that the rules of the road—both formal and informal—in one community may not apply to another.

Peters and Ruff also suggest starting small. For example, tackle a simple bug or documentation fix to start. It will be easier to learn the process and correct mistakes on a small contribution that isn’t critical to your organization’s needs. Companies can and do become major contributors to projects that they didn’t themselves start all the time, but ramping up participation is a process that owes as much to building up relationships at both the individual and overall community levels as it does to cranking out code.

Creating an Open Source Program Office

Increasingly, the logical next step for many organizations is to formalize their participation in open source by establishing an open source program office (OSPO).

Nithya Ruff and Duane O’Brien identify a number of common problems that tend to indicate establishing an OSPO could be useful.¹⁰ Several have to do with a general lack of institutional knowledge about participating in open source. Or maybe there’s some knowledge but it’s in isolated pockets; no one is quite sure where to go to get their questions answered. The legal team is getting overwhelmed with ad hoc requests, and it’s hard to get definitive answers about open source “compliance”—or indeed to fully understand what the pertinent issues even are.

More broadly, there may just be a lack of overall strategy. What projects should be of interest? What events or organizations would it benefit the company to sponsor? How do you establish a reputation in open source communities so you can better attract developers? (Participating in open source as a way to attract talent is a motivation that I hear more and more.)

As with open source communities themselves, there’s a lot of variation in OSPOs. Ruff and O’Brien identify 6 Cs: “Communicate, consume, contribute, collaborate, create competency, and comply.” However, they go on to note that some offices will be of narrower scope than others. For example, one office might be primarily focused on reducing risk to the company by ensuring that the use of open source code doesn’t create any compliance issues from a legal perspective.

A different office might be seeking to improve collaboration within the company by using techniques and workflows inspired by open source projects. Tim O’Reilly is credited with coining the term “inner sourcing” to describe this use of open source development techniques within the corporation.

In a pattern that’s probably becoming familiar in the context of open source communities and projects, the authors of the Linux Foundation’s “Creating an Open Source Program” write that “For every company, the role of the open source program office will likely be custom-configured, based on its business, products, and goals. There is no broad template for building an open source program that applies across all industries—or even across all companies in a single industry. That can make its creation a challenge, but you can learn lessons from other companies and bring them together to fit your own organization’s requirements.”¹¹

The purpose will also influence where the office is hosted within a company. If there’s a strong focus on mitigating legal risk, the legal department might be a

¹⁰<https://osls18.sched.com/event/DjtZ/so-youve-decided-you-need-an-open-source-program-office-duane-obrien-indeedcom-nithya-ruff-comcast>

¹¹<https://www.linuxfoundation.org/creating-an-open-source-program/>

logical fit. If, on the other hand, the emphasis is more on participating in open source communities, that seems a better match with the CTO’s office or elsewhere in the engineering organization.

The leadership of the OSPO can be influenced by these considerations as well. If the motivations are largely driven by internal aspects such as improving collaboration, then it may be important to pick someone who knows how to navigate the twisty byways of a typical enterprise organization. If you’re more outwardly facing—perhaps you want to get involved in some existing open source projects that relate to your company’s strategic interest—you may be better off with someone who has experience and skills in open source communities and practices.

Ruff and O’Brien identify good traits to have as including the following: consensus builder, some technical skills, project management, community roots, and presentation chops. It’s probably obvious from that list that OSPO leadership is heavily weighted to connecting, influencing, and persuading.

That because, as Jeff McAffer, director of the Open Source Programs Office at Microsoft, writes: “This is a culture change endeavor. The code is obviously a big part of it, but the community and the engagement is a people-to-people thing. If you’re going to start an open source program office and you’re going to try to make it a real thing, you’re going to need to understand the culture and get somebody in place who can help drive that culture to a new level. Your head of open source is really a change agent.”

The Water Is Fine

Open source licensing, compliance, and participation can be complicated topics. Understanding all the nuances of any of them can require years of study and experience (and you still won’t always get it right). But that’s true of most things.

In fact, developing an appreciation for the broad brushstrokes that make up open source software’s foundations, legal framework, and community model goes a long way toward being in a position to participate productively. You won’t know—and aren’t expected to know—everything at first. But you can start asking the right questions, finding the appropriate on-ramps, and realizing where you need to hire or develop expertise.

Those are good initial steps to taking advantage of the open source development model and starting to appropriately apply open source approaches to different aspects of your business.

CHAPTER 3

Open Source Development Model

We’ve seen some of the ways in which the thinking around free and open source software has evolved over time. Perhaps the biggest change is the realization that it can be such an effective development model as opposed to just a source of user freedoms. As a result, understanding modern open source has to include understanding how it’s developed. And more specifically, how it’s developed when multiple individuals and organizations participate and collaborate.

This chapter delves into the open source development model, which is best thought of as a toolbox of practices that need to be tailored to the requirements of a given community and project. There’s no single governance model. Projects have different goals and missions. Communities develop across a variety of lines based on their membership and what they’re trying to accomplish.

Open Source Is About Development

Although it’s not necessarily the best lens through which to view modern open source development, Eric Raymond’s *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* (O’Reilly Media, 2001), an essay and later book, is frequently cited. First published in 1997—about the same time that the “open source” term was finding its way into the wild—it contrasted code developed by an exclusive group of developers with code developed in public over the Internet. This was also around the earliest point in time at which you could really talk about the “public” having access to the Internet rather than the employees of a fairly small number of companies or researchers at elite universities.

Central versus Distributed Control

Raymond took Linux as his model for the bazaar. He writes that “Linus’s innovation wasn’t so much in doing quick-turnaround releases incorporating lots of user feedback (something like this had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it wasn’t unknown for him to release a new kernel more than once a day! Because he cultivated his base of co-developers and leveraged the Internet for collaboration harder than anyone else, this worked.”

Today, *The Cathedral and the Bazaar* is often recalled as a commentary on proprietary versus open source software.

The cathedral is the architected and carefully constructed creation of an insular group of specialist trades. Its design the result of singular vision and skilled craft. A tidy and managed process leading to an inexorable endpoint. The result may or may not be beautiful but it will be deliberate.

By contrast, the bazaar is messy. Anyone can participate. There’s duplication. It’s not efficient. Some of the stalls will have higher quality goods than others. The whole evolves organically. The bazaar encourages broad participation and experimentation, which may be ultimately effective for the customers. But you need to accept that it won’t be neat and organized and that no one is going to necessarily going to be able to step in and exert ultimate control.

Raymond was indeed writing about two different software development approaches, but he was actually contrasting two different types of *free software* development practices. Specifically, he was comparing the cathedral-like development of programs like Stallman’s GCC (The GNU Project’s C language compiler)—which he argued was stagnating—with the more bazaar-like development of Linux with its diverse population of contributors.

That said, looking back at *The Cathedral and the Bazaar* with the advantage of 20 years of hindsight, the bazaar remains a useful lens through which to view the often unruly open source development landscape.

Differing Open Source Approaches

Open source software development does tend toward the bazaar rather than the cathedral. However, as even Raymond acknowledged at the time: “. . . There is a more fundamental error in the implicit assumption that the cathedral model (or the bazaar

model, or any other kind of management structure) can somehow make innovation happen reliably. This is nonsense. Gangs don't have breakthrough insights—even volunteer groups of bazaar anarchists are usually incapable of genuine originality, let alone corporate committees of people with a survival stake in some status quo ante. Insight comes from individuals. The most their surrounding social machinery can ever hope to do is to be responsive to breakthrough insights—to nourish and reward and rigorously test them instead of squashing them.”

As we consider the open source development model and the communities associated with it, these points about insights, and hence innovation, are the ones we should most take away from Raymond's essay. Successful projects and communities have a bit of cathedral *and* a bit of bazaar in them.

A Caveat

At this point, it's worth noting that not all open source software really uses an open source development model.

Many of the projects in a public repository like GitHub are occasional hobbies or just something that a developer threw together to solve some narrow problem. Even significant projects don't all have broad participation. This can be a problem when many users depend on a project but it doesn't have sufficient funding—as we saw in the case of OpenSSL.

One company may also drive a project that never attracts significant outside participation for whatever reason. Such projects may be successful by reasonable metrics. They may serve as upstream projects for successful commercial products. They just don't really benefit from the open source development process and probably don't operate in a way that looks much like an open source project with a diverse set of participants. They therefore aren't a good study point for how open source software development is different.

But open source as a strategic approach to software development follows certain models and takes certain approaches, and that's the topic for this chapter.

Models for Governing Projects

The open source licenses that we covered previously provide some basic legal ground rules for a project and the community that forms around that project. However, licenses don't speak to how decisions are made, how members will interact, what processes

the community will follow, and how it will be structured. These fall into the domain of governance. You can think of governance as a social and decision-making framework for a project.

The work “framework” is important. An explicit project and community governance model can help to provide a general approach to moving a project forward. It makes key assumptions explicit. However, as I’ll discuss later, there’s much more to encouraging new contributors, maintaining an existing community, and dealing with conflicts that have to take place whatever the governance model in place.

At a minimum, a framework should capture the following:

- How decisions are made;
- The core principles of the project;
- The core goals of the project;
- The project’s community.

Scott Nicholas, senior director of Strategic Programs at the Linux Foundation, advises that certain decisions are best addressed up front. In particular, he highlights license and IP decisions, mission and scope, and starting technical roles. At the same time, he also recommends that less is more, and it’s often best to “empower decision making by the various project bodies as opposed to attempting to address all potential decisions up front.” For example, oversight bodies can set their own rules for voting procedures and elections while technical bodies can set their own technical roles and project life cycle guidelines and procedures.

Who Decides?

If you’ve ever been involved in setting up an organization with any sort of formal structure—or even just tried to get a large group to decide on where to go for lunch—you know how difficult decision making can be absent a process.

But it’s not even so much that making routine decisions is hard. Convivial groups can usually reach consensus without *too* much difficulty when the stakes are low and there’s broad agreement on facts and criteria. Then the day comes when something polarizing comes along and there’s no playbook to make the controversial call. My own experience as a board member for a small non-profit is that a casual approach to the decision-making process works well. Until the day it doesn’t.

Which isn't much different from decision making in many healthy organizations. When there are irreconcilable differences, it matters how the logjam gets broken. Some projects do indeed have stricter hierarchies than others. But many of the same principles apply to making decisions in open source communities as elsewhere.

There are many ways to think about differences among open source communities. One can distinguish among single vendor open source projects, development communities, user communities, and open source competence centers—although the lines between these aren't hard and fast.

For example, single vendor open source projects suggest projects that remain under the control of a single software company. However, these can range from projects that are really open source in name only to projects that, at least in principle, welcome outside contributors, but a company retains ultimate decision-making authority over the core project.

Communities of developers and communities of users often blend together. In fact, it's been something of a trend for major open source foundations and communities like OpenShift Commons to increasingly be as much about bringing together end users with each other and with developers as more traditional communities that were mostly focused on writing code.

As for open source competence centers within enterprises, those can start to get hard to distinguish from open source program offices at vendors as the lines between who is a software company and who isn't blur.

It's worth observing that, even though governance of open source projects is important, it's not always explicit. Researcher Javier Canovas studies how software is developed. He queried the widely used GitHub software repository in 2018 to collect the 25 most-starred—think of this as roughly equivalent to most popular—open source projects on the platform.¹ He found that about three-quarters didn't provide any information about the governance model used by the project.

There were exceptions. For example, he found that “the Node.JS project provides a detailed description of the main developer roles of the project, their responsibilities, and how they are elected.” (It's perhaps worth noting that this project is part of a foundation, the Node.JS Foundation, which tend to be associated with more formal governance structures.) But, although the situation was better than it was when he last looked at the numbers two years previously, a lot of significant projects still aren't explicit about how

¹<https://opensource.com/open-organization/18/4/new-governance-model-research>

they were run. Lack of license information on GitHub has also been an issue in the past although it now takes more of a deliberate effort to avoid choosing a license and the percentage of projects with a license has risen correspondingly.

Community projects generally fall into three broad buckets with respect to basic decision-making approaches.

Benevolent Dictator for Life

There's the BDFL (Benevolent Dictator for Life). One person, who is usually the project's founder, generally has final say on major project decisions. This is the norm for smaller projects, which may not even have a formal governance process. However, many major open source programming languages also take this approach. And, notably, Linux—while having well-established development processes and formal structure—also has Linus Torvalds as a BDFL by most reasonable definitions.

Projects that are under the direct control of a single company or other organization often use a process that looks similar in practice. Outside contributions may be encouraged and decision making delegated to proven contributors. However, the right to make decisions about fundamental project direction and similarly important matters are often retained by the commercial entity in ultimate control.

My Red Hat colleague, Dave Neary, notes that BFDL control is often *de facto* rather than structured in a formal manner.

He adds that “there are a number of reasons why it's hard for companies to get involved when one vendor is dominant; it is like jumping on a train that is speeding through a station. The learning curve is steep and the community is not set up to onboard new ‘outside’ contributors. Another reason is an organizational fear of being a second class citizen; until you have a 2nd company engaging, and can see how they are treated, decision makers figuring out whether to engage or not assume the worst.”

I include a benevolent dictator model because many projects end up here by default. However, as another colleague, Joe Brockmeier, observed to me, it's mostly considered an anti-pattern at this point. “While a few BFLD-driven projects have succeeded and continue to do well, others have stumbled with that approach,” he says. On the one hand, the BFDL model means that there is a clear, unambiguous decision maker. On the other hand, ego, and conflicts of interest between a project and commercial ambitions can easily drive a wedge between the community and an increasingly malevolent dictator—causing the project to flounder or fork.

Meritocracy

Meritocracy as a formal approach is most associated with the Apache Foundation. Unlike Linux, Apache's original project, the Apache web server, was started by a diverse group of people with common interests rather than by a single developer. As the project developed, in the words of the Apache Foundation, "when the group felt that the person had 'earned' the merit to be part of the development community, they [were] granted direct access to the code repository, thus increasing the group and increasing the ability of the group to develop the program, and to maintain and develop it more effectively. We call this basic principle 'meritocracy': literally, government by merit." Apache has a formal structure built on these principles, which goes so far as to require that contributions be only made by individuals representing themselves rather than their employer. While projects outside the Apache Foundation may not have such a formal structure and approach to meritocracy, the basic idea of meritocracy is a common principle and practice.

As a side note, the "meritocracy" term has a controversial, if somewhat obscure, history. British sociologist and politician Michael Young published *The Rise of the Meritocracy* in 1958 (Thames and Hudson, 1958). In this book, he described a future dystopian society stratified between an intelligent and merited power holding elite and a disenfranchised underclass of the less merited. However, in the tech industry, meritocracy is mostly used in a positive way to connote decisions ostensibly made on the basis of knowledge and experience rather than rank or financial contribution.

Consensus

Under a consensus, sometimes called a liberal contribution, model, Open Source Guides writes that "the people who do the most work are recognized as most influential, but this is based on current work and not historic contributions. Major project decisions are made based on a consensus seeking process (discuss major grievances) rather than pure vote, and strive to include as many community perspectives as possible."²

In practice, there are fewer differences in how decisions get made in healthy communities than the categories may indicate. For example, the meritocratic Apache Foundation writes about how the process to resolve negative votes on proposals "is called 'consensus gathering' and we consider it a very important indication of a

²<https://opensource.guide/>

healthy community.” Conversely, even if decisions are always made by consensus, this inherently suggests a fallback to some sort of majority rules mechanism for the times when everyone can’t come to an agreement.

What Are the Principles?

Some of the foundational principles common to many open source projects are fairly straightforward. For example, projects should generally use an approved OSI license. Projects should also have a clear statement around intellectual property. For example, all contributions to projects under the Eclipse Foundation’s top-level project charter must adhere to the Eclipse Foundation Intellectual Property Policy.

Open First

Less obvious for companies accustomed to developing more traditional software is the need to work in the open. Take the example of Kubernetes. It started out as an internal Google project that can be traced back to Borg, Google’s internal container cluster management system. Launched in 2014, Kubernetes emerged as the leading technology to deploy and orchestrate containers for so-called cloud-native workloads.

In 2016, Google turned Kubernetes over to the Cloud Native Computing Foundation (CNCF). As CNCF executive director Dan Kohn told me, “one of the things they realized very early on is that a project with a neutral home is always going to achieve a higher level of collaboration. They really wanted to find a home for it where a number of different companies could participate.”

However, when Sarah Novotny took over as program manager of the Kubernetes community at Google, she found shifting the company mindset toward open participation took work.

In 2017, she said that “giving up control isn’t always easy even when people buy into doing what is best for a project. Defaulting to public may not be either natural or comfortable. ‘Early on, my first six, eight, or 12 weeks at Google, I think half my electrons in email were spent on: Why is this discussion not happening on a public mailing list? Is there a reason that this is specific to GKE [Google Kubernetes Engine]? No, there’s not a reason.’ There were lots and lots of conversations that looked like that initially just to remind the Googlers that by default they should be discussing this publicly if they wanted the transparency, and the openness, and the engagement from the community that intellectually they did.”

Separating Technical and Business Decisions

Another core principle for many projects is the separation of business and technical governance.

As Red Hat's Jason Baker writes in the case of OpenStack: "The OpenStack Foundation has roles for many different types of contributors in their structure. The Board of Directors oversees financial decisions and long-term strategy, while the technical committee—not surprisingly—gives technical direction, and the user committee helps ensure the project is meeting the needs of organizations working with the software on the ground."³

This gets to the ideal, which finds its way into practice to greater or lesser degrees, that open source projects should be steered, at least in part, by those contributing the code. This is a fundamentally different approach from the marketing textbook view of product development in which product requirements are gathered, business priorities assessed and, only then, development priorities established. There is a broader and more diverse set of stakeholders in open source software development than with proprietary software, which is certainly a contributor to open source's often bazaar-like atmosphere.

Or take the Fedora Project, which creates Fedora—a Linux distribution intended to focus on innovation and integrate new technologies early on that also serves as a community upstream project for Red Hat Enterprise Linux. Its top-level community leadership and governance body, the Fedora Council, is composed of a mix of representatives from different areas of the project, named roles appointed by Red Hat, and a variable number of seats connected to medium-term project goals. Most of the project is then roughly organized under the FESCo (the Fedora Engineering Steering Committee) or the Mindshare Committee.

Best Practices for Setting Goals

Setting goals is about alignment. What's the project's mission? What's the project's scope? What's not in scope?

Projects and communities differ. Novotny talks about how different projects "have different goals as an open source project, and each of them have different needs, challenges, and opportunities. Every one of them says, 'Oh, I don't want to be like this

³<https://opensource.com/business/14/4/governance-openstack>

project,’ or ‘Oh, I want to be like this project,’ but they only mean a subset, so you build these independent cultures in each of these cases.”

It’s instructive to see what setting goals looks like in the context of a specific project, in this case the aforementioned Fedora Project, which adopted a new mission statement in 2017.⁴

As Matthew Miller, the Fedora Project leader, wrote on the Fedora mailing list: “Way back in 2003, the original Fedora Project mission statement was straightforward—‘to work with the Linux community to build a complete, general purpose operating system exclusively from open source software.’ This has some virtue: it’s clear and concrete, and it encodes the values of community and open source. But, it’s also rather small; arguably this was *_already done_*, so, you know, ‘good job everyone’ — backslapping ensues, nothing more needed, right?”

Miller went on to write that “We wanted something which would answer: What do we do?; How do we do it?; Who do we do it for?; and, what unique value do we bring?” What the Fedora Council came up with was this: “Fedora creates an innovative platform that lights up hardware, clouds, and containers for software developers and community members to build tailored solutions for their users.”

A few things here are particularly worth highlighting.

- **Create an innovative platform:** It’s about focusing on new things, not just integration. It incorporates the latest technology, not just technology that is proven and safe for critical workloads. (This hearkens back to the earlier projects versus products discussion.)
- **Lights up hardware, clouds, and containers:** Related to the above, this phrase suggests a focus on the new types of environments on which users want to run software.
- **For software developers to build tailored solutions for their users:** This speaks to Fedora as a platform for developers who want to build on top of it.
- **For community members to build tailored solutions for their users:** As Miller notes, “Many of our contributors are here to collaborate to create solutions for specific user problems, ranging from Fedora Workstation to Fedora Robotics Suite.”

⁴<https://lwn.net/Articles/720055/>

Fedora also lays out what it calls “Four Foundations”: Freedom, Friends, Features, and First, core principles that were not changed when it came up with its new mission statement.⁵

Who Is in the Community?

The simplest (but also simplistic) way to think about the community is through the lens of code contributions. The leader, maintainer, committer, and contributor taxonomy is fairly typical.

Leaders

Someone or someones need the authority to make final decisions about features, releases, and other activities. This may be a “benevolent dictator for life,” a technical oversight committee, or some other voting group that has the explicit authority to make technical decisions.

Maintainers

Most projects delegate some of the decisions to people who are responsible for maintaining specific parts of the project, and in large projects, these maintainers may also delegate to people who are responsible for subcomponents of their portion. For example, Greg Kroah-Hartman is the current maintainer for the Linux kernel stable branch as well as for a variety of subsystems.

In a large project, maintainers can be more editors than coders. For example, Kroah-Hartman says that most of his work involves code review of the latest development kernel. He also talks about working with developers to help them learn how to be part of the Linux kernel development community, engagement that’s often not all that technical in nature.⁶

In fact, in *Producing Open Source Software: How to Run a Successful Free Software Project* (O’Reilly Media, 2005), Karl Fogel notes that not all maintainers need be coders at all. He writes: “There may be people who are very invested, and who contribute a lot, but through means other than coding. Plenty of people provide legal help, or organize events, or manage the bug tracker, or write documentation, or do any number of other

⁵<https://docs.fedoraproject.org/fedora-project/project/fedora-overview.html>

⁶<https://thenewstack.io/greg-kroah-hartman-commander-chief-linux-stable-branch/>

things that are highly valued in the project. They often may have a level of influence in the community (and familiarity with the community's dynamics) that exceeds that of many committers."⁷

Committers

In many projects, maintainers and committers are the same thing. However, especially in large projects, there's often a broader group of developers who are considered reliable and responsible enough to be allowed to directly submit code into the project's version control system without having to go through a maintainer gatekeeper. Such contributions are still subject to review by maintainers or project leaders and may be reverted if there is a problem or processes weren't followed. For example, many projects require that code pass an automated test suite before it can be added to the project and code that doesn't will be rejected.

Fogel argues that "The most important criterion is not technical skill or even deep familiarity with the code, but rather that a person shows good judgement. Judgement includes knowing what not to take on. Someone might post only small patches, fixing fairly simple problems in the code, but if their patches apply cleanly, do not contain bugs, and are mostly in accord with the project's log message and coding conventions, and there are enough patches to show a clear pattern, then an existing committer should propose them for commit access."

Contributors

Healthy projects also typically have a much broader pool of people who contribute in smaller ways. They may make a small bug fix or spot an error in the documentation. Their contributions are usually subject to a review from an experienced committer or maintainer, but the breadth of this group is often one of the hallmarks of broad support for an open source software project.

Providing a gentle on-ramp for new contributors and providing mentoring and other means to get them incrementally more involved in the project as they desire is one of the most important duties of community management.

⁷<https://producingoss.com/>

Peters and Ruff offer guidelines for contributors in “Participating in Open Source Communities.”⁸

They suggest that if you are a first-time contributor to a project, you might consider finding a mentor or an experienced project member who can review your work and provide you with some feedback as you prepare your first couple of contributions.

After submitting a contribution using the process described in the documentation, you will need to be available to respond to feedback. Common feedback would include questions about how something works or why you chose a particular approach along with suggestions for improvements or requests for changes. This feedback can be tough sometimes, so it helps to assume that the feedback is in the spirit of making your contribution better and avoid getting defensive. You may need to go through several rounds of resubmission and additional feedback before your code is accepted, and in some cases it may be rejected. There are many valid reasons why something might not be accepted, so don’t take it personally if your code is rejected; and if possible, try to learn more about why your contribution was not accepted to help increase the chances of getting your next contribution included.

Keep in mind that if your contribution was accepted, you may be expected to maintain it over the long term. This is especially true for large contributions, new features, or stand-alone code, like a driver for a specific piece of hardware. For small contributions and bug fixes, it is unlikely that there will be any long-term maintenance expected.

Why You Should Think About More Than Coders

Communities aren’t just about coding though—or even about contributing tangible things that are an inherent part of the project like documentation, a website, or logos. But as Diane Mueller, the director of community development for OpenShift Origin, told me, it can be hard to get away from a code-centric mindset. “You only really talked to people when they were those people that were giving you code. We tried to flip this all on its head” with OpenShift Commons, she says.

She adds that “In addition to all these people who were adding services, or providing infrastructure, or working with us on this, there was that whole other community out there, the customers, the people who were actually deploying OpenShift Origin or

⁸<https://www.linuxfoundation.org/participating-open-source-communities/> Creative Commons Attribution ShareAlike 4.0 International License

deploying OpenShift Container Platform. How do we get their feedback back to the contributors, the engineers, the service providers on this topic? What we did was try and create a new model, a new ecosystem that incorporated all of those different parties, and different perspectives . . . A lot of it was about creating this peer-to-peer network model, wherein Red Hat got out of the way so the conversations could be common between Amadeus and a Clydesdale Bank or someone else.”

Users Get Involved

The direct involvement of software users in the open source development process isn’t new. As we’ve seen, users wrote the first operating systems.

AMQP (Advanced Message Queueing Protocol) is another great study point. John O’Hara of JPMorgan Chase conceived of it in 2003 as a cooperative open effort. The company designed it over the next couple of years. In 2005 JPMorgan Chase approached other firms to form a working group that included Cisco Systems, IONA Technologies, iMatix, Red Hat, and Transaction Workflow Innovation Standards Team (TWIST). Today, implementations of AMQP are widely used in financial services and elsewhere where high-performance middleware messaging is needed.

However, the broad involvement by many types of end users in setting the direction of open source projects and working cooperatively on software that’s relevant to entire industries has increased by such a degree as to be a difference in kind. As with open source development carried out cooperatively by otherwise competing vendors, legal and other barriers to cooperation among users of the software are also lowered by open source.

Users Become Contributors

Users also can become contributors. Fogel observes that “Each interaction with a user is an opportunity to get a new participant. When a user takes the time to post to one of the project’s mailing lists, or to file a bug report, she has already tagged herself as having more potential for involvement than most users (from whom the project will never hear at all). Follow up on that potential: if she described a bug, thank her for the report and ask her if she wants to try fixing it. If she wrote to say that an important question was missing from the FAQ, or that the program’s documentation was deficient in some way, then freely acknowledge the problem (assuming it really exists) and ask if she’s

interested in writing the missing material herself. Naturally, much of the time the user will demur. But it doesn't cost much to ask, and every time you do, it reminds the other listeners in that forum that getting involved in the project is something anyone can do."

How to Encourage New Contributors

In writing this section, I received a lot of great advice from my colleagues in Red Hat's Open Source and Standards group who think about issues of community and open source contribution a lot. That's because one of the most challenging jobs of a community manager is growing the breadth of participation in their project.

This conversion of users into contributors is one important aspect of community management. Mind you. There's no rule that users always start contributing code or anything else. But successful open source software projects have an on-ramp that makes it easy for casual or occasional contributors to begin participating on their own terms—while recognizing that others will be in a position to fully participate in a project at a high level from Day One.

Projects tend to start as an in-group, whether employees of a single company or colleagues of another sort. This group may desire more outside participation in principle. But that generalized wish may not extend to making an explicit effort to be welcoming to outsiders, relinquishing control, or abandoning processes that cordon off decision making from newcomers.

Holding Onto Control: An Anti-Pattern

These behaviors are natural but they're also anti-patterns. Dave Neary writes that: "Companies are used to controlling the products they work on. Attempting to transfer this control to a project when you want to grow a developer community will result in a lukewarm response from people who don't want to be second class citizens. Similarly, engaging with a community project where you will have no control over decisions is challenging. Exchange control for influence."⁹

Neary also argues that the hesitancy to give up control can be part of a broader lack of understanding of the role of community in which the community is viewed through

⁹<https://www.slashdata.co/blog/2011/01/open-source-community-building-a-guide-to-getting-it-right>

the lens of a resource that does stuff for you for free. Companies just expect other people to do work for them as directed. Neary writes of “the technical director who does not understand why community projects do not accept features his team has spent months developing, or the management team that expects substantial contributions from outside the company to arrive overnight when they release software they’ve developed.”

Tidelift’s Chris Grams has described this as the Tom Sawyer model of community engagement. He writes that “I’ve talked to people at companies who are considering ‘open sourcing’ their product because they think there is an army of people out there who will jump at the chance to build their products for them. Many of these people go on to learn tough but valuable lessons in building community. It’s not that simple. You can avoid the Tom Sawyer trap through some exercises in humility. A couple of questions for you: Why does it have to be your community and why do you have to build it? Do people have to come participate in your community, or might you consider joining their community?”¹⁰

As a general principle, think about the friction points that might inhibit someone, especially someone from outside your organization, from contributing to the project. I’ve already touched on some of these. Certain licenses or contributor license agreements may be sensible from a legal and intellectual property perspective. Nonetheless, be aware that the choices you make there can affect how likely someone is to want to work on the project.

Tools also play a role.

Reducing the Friction of Tools

Amye Scavarda of the Gluster storage project talks about how infrastructure was historically hard to open up to outsiders without a track record but that “infrastructure tools like Ansible, Chef, and Puppet have become widely adopted and changed this. It’s now possible to open source project infrastructure as code, with the same levels of access as any other contributor. This makes the process visible to contributors, allowing a pathway for contribution that might not be strict project code. You’re no longer tied to the problem of only a limited amount of high-level contributors who have access.”¹¹

¹⁰<https://chrisgrams.com/2009/09/09/tom-sawyer-whitewashing-fences-and-building-communities-online/>

¹¹<https://community.redhat.com/blog/2017/04/encouraging-new-contributors/>

More generally, both virtual machine- and container-based technologies simplify setting up replicable development environments quickly.

Conversely, obscure and difficult-to-use version control systems, oddball programming languages, complex procedures, hard-to-run test suites, and a litany of other frictions make it easy for a potential new contributor to quickly decide that “it’s just not worth it” and move on. Very few organizations erect these roadblocks on purpose. Maybe their engineers have gotten so used to convoluted legacy processes and tooling that it doesn’t hurt so much any longer. But the more familiar you can make working with the code base (or otherwise contributing to the project), the more contributors you’ll gain.

Mentoring

Formal mentoring programs can also help to get contributors up to speed. Examples include the Kubernetes Mentoring Initiatives and OpenStack Mentoring. The details vary. For example, OpenStack offers two different levels of mentoring in addition to a long-term mentoring program that’s longer term, but more lightweight, and is aimed at bringing together mentors and mentees who may not be co-located.

Neary observes that mentoring programs aren’t always effective because of communication issues, excessive time commitments, and lack of follow-through.¹² He nonetheless argues that “Mentoring programs are needed to ensure that your project is long-term sustainable.”

He likens mentoring to management and suggests going for quick wins while developing an ongoing relationship between mentors and “apprentices.” He cautions that “Just as not everyone is suited to being a manager, not everyone is suited to being a mentor. The skills needed to be a good mentor are also those of a good manager—keeping track of someone else’s activity while making them feel responsible for what they’re working on, communicating well and frequently, and reading between the lines to diagnose problems before it’s too late to do anything about them.”

However, he adds that “Mentoring is also an opportunity for developers who would like to gain management experience to do so as a mentor in a free software project.”

¹²<https://blogs.gnome.org/bolsh/2011/05/31/effective-mentoring-programs/>

The Importance of Culture

More broadly though, communities need to think about their culture. I was going to write here that communities need a healthy culture in order to attract new contributors. But that's (unfortunately) not quite right. Some projects are connected to sufficiently important commercial products that they can be successful in attracting developers almost in spite of themselves. Nonetheless, there's been enough repudiation of bad behaviors and lack of diversity in various open source projects that we can hopefully take cultures that value attributes like empathy, respect, and inclusiveness as aspirational even if they aren't always the norm.

Jono Bacon refers to the art of community as “building belonging.” He writes that “A sense of belonging is what keeps people in communities. This belonging is the goal of community building. The hallmark of a strong community is when its members feel that they belong. Belonging is the measure of a strong social economy. This economy's currency is not the money that you find in your wallet or down the back of your couch, but is social capital. For an economy and community to be successful, the participants need to believe in it. If no one believes in the community that brings them together, it fails.”¹³

Structures and processes matter when welcoming contributors and maintaining communities more generally. However, at least as important are the intrinsic attitudes, norms, and culture of the community.

Steps to Maintain a Community

Part of maintaining a community is encouraging new contributions. Contributors leave projects all the time, and they don't necessarily leave because someone did something wrong. They may just move onto new things, need a break, or otherwise want to try something different. This means that the community needs to be constantly refreshed with new blood. It's therefore important that new contributors have a good experience and that community members be able to progress according to their interests.

¹³<https://www.jonobacon.com/2011/05/31/the-art-of-community-building-belonging/>

Quick Responses

Josh Berkus thinks of making a contribution as akin to starting a timer.¹⁴ He writes that: “The longer maintainers take to respond to their submission, the lower the chance that person will ever contribute to the project again. While no one has done a specific study on this that I know of, my experience is that it drops by half in a couple of days and it gets very close to zero in only a few weeks. If you’re running a young project and looking to attract lots of new contributors, there may be no higher priority than dealing with first-time contributions quickly. Minutes to hours is the target here.”

It’s not just the initial response that matters though. It’s certainly a good thing to quickly acknowledge a contribution. But it’s not enough if the contribution never gets added to the project for reasons that aren’t ever made clear. This gets back to the importance of infrastructure that Scavarda discussed earlier. Automated tooling and continuous integration/continuous delivery (CI/CD) systems can reduce the time to integrate and test new software. It doesn’t eliminate the need for someone to work with a new contributor when, as is often the case, their submittal needs some additional work. But it does reduce a lot of the latency associated with manual processes.

Documentation: An Easy On-Ramp

Berkus adds that documentation is another good area to automate integration given that “it’s one area where most open source projects need a lot more contributions and where acceptance of imperfect submissions carries relatively low risk.”

The design of the software itself also plays into how people can engage with a project. For example, writing in 2013, Simon Phipps discussed some learnings from the LibreOffice project.¹⁵ “The code base was hard to build, so the project set up automated Tinderbox continuous-integration build servers, allowing any developer to work on the code without needing to create their own complex build environment in multiple operating system environments. The code has been substantially cleaned up with translation of comments from German to English for more accessibility around the world (most developers have English as at least a second language). The clean-up also

¹⁴<https://community.redhat.com/blog/2017/04/contributors-speed-matters/>

¹⁵<https://www.infoworld.com/article/2613624/open-source-software/what-you-can-learn-from-the-monster-libreoffice-project.html>

involved a great deal of refactoring of old approaches into more modern ones and the elimination of unused code left over from defunct platforms—this is 20-year-old code, after all.”

Modular Beats Monoliths

Greg DeKoenigsberg and Michael DeHaan talk about how modularity and “option value” have helped make the Ansible project successful, quoting from a 2005 paper by Carliss Baldwin and Kim Clark of Harvard Business School entitled “The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model?”

Modularity is important because it “provides a simple framework of platform and modules. The platform supports the modules and provides well-defined rules for module development; modules may then be developed or modified independently according to those rules, thus allowing contributors to add value in corners of the project, with minimum investment.”

For its part, high option value means users can choose some modules but not others, or even to rewrite particular modules.

Combining the two makes it easier to identify ways of contributing immediately, and to see the value of that contribution quickly. “In browsing through Ansible’s libraries, one can find over 230 different modules, the vast majority of which were co-developed and co-maintained by multiple community developers over time. The Ansible ‘batteries included’ philosophy helps to ensure that as new modules are developed, they are tested and integrated in Ansible’s core, so that the maximum number of users get the maximum option value for the minimum collective effort,” DeKoenigsberg and DeHaan write.

Communicate, Communicate, Communicate

Effective communications are important for any team and not just an open source community. However, many open source communities bring to the fore considerations that haven’t historically been as important for teams working together in a single location.

The Limits of Being Together

In a traditional work setting, a lot of communications can take place fairly organically. You hold impromptu meetings, bump into coworkers in the hallway, eat lunch together, and make decisions by the water cooler. Communications can almost be thought of as being embedded into the physical nature of the workplace. Indeed, we see this idea of collaboration through physical proximity as a stated benefit of the much-maligned open office plans that are so in vogue.

Not that co-location is necessarily a panacea. Recalling one of her teams in which everyone sat next to each other, Red Hat chief Agile architect Jen Krieger observed to me that “we didn’t necessarily want to be in everybody’s face all the time. If you are an engineer, it’s really sometimes hard to be in a situation where you’re constantly pulled out of what you’re trying to think about. We would actually pick a day every week where everybody would be working from home, so that folks could focus on just getting their work done.”

Best Practices for Distributed Teams

But even the best traditional practices break down with widely distributed communities. What may be less obvious is that they start to break down when communities are distributed to even a small degree. What happens is that you talk about software development or something else entirely with the local team over pizzas and beers and you make some collective decision. Then the two or three guys or gals in London or Hawaii only find out after the fact, perhaps completely by accident. “That’s right, we made that decision over lunch last week. Sorry. We’ll try and remember to tell you next time.”

Brockmeier notes that situations such as this are the reason that the Apache Software Foundation has the “if it didn’t happen on the mailing list, it didn’t happen rule.” He says that “Any major decision that wasn’t discussed on a mailing list could very well be reversed. ‘Major decision’ is very much up for interpretation, but I’ve seen a number of things backed out and put up for discussion because they violated this rule.”

A partially distributed team can be the most challenging. Krieger tells a story about how she was the sole distributed team member of a team trying Scrum (an Agile software development technique) for the first time. “Everybody else was in our Miami office and they were all, on a daily basis, participating in live and active vocal conversation . . . A lot of times, what would happen is they would have team meetings and they would forget to

call me, because the phone exchange didn't allow direct call in, if they didn't call me, I just didn't go to the meetings."

She adds that water cooler conversations are going to happen in any case but "when you are making decisions, the first thing I always like to tell people is, 'Who you actually bring to the decision making room is actually a pretty important topic.' If you are going to randomly make decisions in the hallway without all the people who need to be in that decision, it's probably not going to be a very good way for you to actually get consensus or drive change to your organization." Which is good advice for both distributed and local teams.

It's About People

Finally, Krieger advises that you think about team and community dynamics rather than purely mechanical processes around information exchange and decision making. For example, she says that "It's important for teams to understand, especially if you're committed to doing something together as a team, that you have a social responsibility to the folks that you're actually interacting with. It's not just, get on your team meeting and immediately start talking about work but it's asking somebody about their day, or finding out what they did over the weekend, or trying to make some sort of social connection so that it's easier for you to have casual conversation."

There are a wide range of tools to make remote collaboration easier.

We call IRC and Slack messaging tools, but you can almost think of them as ambient distributed conversation that allows people to participate mostly when they have the time and feel like it. Code repositories like GitHub provide visibility into the software being developed. Shared documents and wikis can be used for agendas and to document decisions. Skype and other video conferencing systems are good for lightweight real-time interactions. Companies can facilitate the rollout of tools and provide guidelines for using them appropriately and consistently. However, Autodesk's Guy Martin suggests an approach that is "culture first, process second, tools last." Good tools are important, but they won't help if the culture and behaviors don't support collaboration.

The Limits of Virtual

That said, even the current generation of collaboration tools lack the richness and serendipity of in-person communications. In setting up OpenShift Commons Diane Mueller said that, while she does a lot of things to try to make the virtual world work, she

also realized that get-togethers couldn't just be virtual. As a result, OpenShift Commons has regular "Gatherings," day-long meetings of a few hundred members from both developer and user communities spread around the world.

Krieger adds that another team she was working on was "having a little bit of trouble actually working well together . . . We brought everybody together so that they could meet each other, have food together. People bond while eating. It's just a thing. It's a universal thing, whether it be beer in the Czech Republic or Italian food in Little Italy in New York City, people bond over eating and drinking. It's just a thing that we do as humans."

In writing about "The importance of face-to-face," Jono Bacon adds that "while there are often attempts in companies to provide functional methods of building relationships (such as those cheesy team-building exercises most of us have suffered through), relationships usually form in more organic ways. Getting to know someone is often a mixture of chatting in the hallway, having breakfast together, sharing ideas at the end of a presentation, having a couple of drinks together in a bar, and other ways of getting to know each other. The venue and proximity are not the only important pieces though. Relationships also typically form when there is a sense that personal barriers have been dropped a little. When people feel comfortable sharing experiences, expressing concerns, riffing on ideas, and confiding in each other, genuine relationships and alliances form."¹⁶

Having said all this, there will always be frictions to communicating within a project and community. Language, time zones, and cultural norms have costs. So does the limited day-to-day communications bandwidth of distributed teams. These all take effort to overcome and, in some cases, the frictions may just be too great. This then comes back to some of the other project characteristics that can make contributing easier. Modularity and bounding the scope of code are also useful for helping individuals and small groups to work in parallel. Communicating still matters, but it's more tolerant of limited real-time, high-bandwidth interactions.

Bacon writes that "Peter Bloch, a consultant on learning, makes an important foundational observation about communication in a social economy: 'community is fundamentally an interdependent human system given form by the conversation it holds with itself.' When I first heard that quote, I realized that the mechanism behind communication in a community is stories. Stories are a medium in which we keep

¹⁶<https://opensource.com/life/15/10/the-importance-of-face-to-face>

the river flowing . . . when the characters in the stories are people in a community, the stories are self-referencing and give the community a sense of reporting. Communities really feel like communities when there is a news wire, be it formalized or through the grapevine.”

Determine If You’re Successful

“Without data, you’re just a person with an opinion.” Those are the words of W. Edwards Deming, the champion of statistical process control who would come to be credited as one of the inspirations for what became known as the Japanese postwar economic miracle of 1950 to 1960. Ironically, Japanese manufacturers like Toyota were far more receptive to Deming’s ideas than General Motors and Ford were.

Community management is certainly an art. It’s about mentoring. It’s about having difficult conversations with people who are hurting the community. It’s about negotiation and compromise. It’s about interacting with other communities. It’s about making connections. In the words of Diane Mueller, it’s about “nurturing conversations.”

However, it’s also about metrics and data.

Some have much in common with software development projects more broadly. Others are more specific to the management of the community itself.

I think of deciding what to measure and how as adhering to five principles.

Measuring Something Changes It

First, you should recognize that behaviors aren’t independent of measurements you choose to highlight.

In 2008, Daniel Ariely published *Predictably Irrational* (HarperCollins, 2008), one of a number of books written around that time that introduced behavioral psychology and behavioral economics to the general public. One memorable quote from that book is the following: “Human beings adjust behavior based on the metrics they’re held against. Anything you measure will impel a person to optimize his score on that metric. What you measure is what you’ll get. Period.”

This shouldn’t be surprising. It’s a finding that’s been repeatedly confirmed by research. It should also be familiar to just about anyone with business experience. It’s certainly not news to anyone in sales management, for example. Base sales reps’ (or their managers’!) bonuses solely on revenue, and they’ll try to discount whatever it takes to

maximize revenue even if it puts margin in the toilet. Conversely, want the sales force to push a new product line—which will probably take extra effort—but skip the spiffs? Probably not happening.

And lest you think I’m unfairly picking on sales, this behavior is pervasive, all the way up to the CEO, as Ariely describes in a 2010 Harvard Business Review article. “CEOs care about stock value because that’s how we measure them. If we want to change what they care about, we should change what we measure,” writes Ariely.

Developers and other community members are not immune.

What Actually Matters?

Second, you need to choose relevant metrics. There’s a lot of folk wisdom floating around about what’s relevant and important that’s not necessarily true. Dave Neary offers an example from baseball. He notes that “In the late '90s, the key measurements that were used to measure batter skill were RBI (Runs Batted In) and batting average (how often a player got on base with a hit, divided by the number of at-bats). The Oakland A’s were the first major league team to recruit based on a different measurement of player performance, on-base percentage. This measures how often they get to first base, regardless of how it happens.”¹⁷

Indeed, the whole revolution of sabermetrics in baseball and elsewhere, popularized in Michael Lewis’ *Moneyball: The Art of Winning an Unfair Game* (W. W. Norton, 2004), often gets talked about in terms of introducing data in a field that historically was more about gut feel and personal experience. But it was also about taking a game that had actually always been fairly numbers-obsessed and coming up with new metrics based on mostly existing data to better measure player value. (The data revolution going on in sports today is more about collecting much more data than was previously available through video and other means.)

Quantity May Not Lead to Quality

As a corollary, collecting lots of tangential but easy-to-capture data isn’t better than just selecting a few measurements you’ve determined are genuinely useful. It’s tempting in a world where online behavior can be tracked with great granularity and displayed in

¹⁷<https://community.redhat.com/blog/2014/07/when-metrics-go-wrong/>

colorful dashboards to be distracted by sheer data volume even when it doesn't deliver any great insight into community health and trajectory.

This may seem like an obvious point. Why would I choose to measure something that isn't relevant? However, in practice, metrics often get chosen because they're easy to measure, not because they're particularly useful. They tend to be more about inputs than outputs. The number of developers. The number of forum posts. The number of commits. Collectively, measures like this often get called vanity metrics. They're ubiquitous but most people involved with community management don't think much of them.

Number of downloads may be the worst of the bunch. It's true that, at some level, they're an indication of interest in a project. That's something. But it's sufficiently distant from actively using the project, much less engaging with the project deeply, that it's hard to view downloads as a very useful number.

Is there any harm in these vanity metrics? Maybe? Yes, to the degree that you start thinking that they're something to base action on. Probably more seriously, they can come to be seen by stakeholders like company management or industry observers as meaningful indicators of project health.

What Does the Number Mean?

Understand what measurements really mean and how they relate to each other. Neary makes this point to caution against myopia. He tells the story of how “in one project I worked on, some people were concerned about a recent spike in the number of bug reports coming in because it seemed like the project must have serious quality issues to resolve; however, when we looked at the numbers, it turned out that many of the bugs were coming in because a large company had recently started using the project. The increase in bug reports was actually a proxy for a big influx of new users, which was a good thing.”

In practice, you often *have* to measure through proxies. This isn't an inherent problem, but the further remove you get from what you want to measure and what you're actually measuring, the harder it is to connect the dots. It's fine to track progress in closing bugs, writing code, and adding new features. However, those don't necessarily correlate with how happy users are or whether the project is doing a good job of working toward its long-term objectives.

Horses for Courses

Finally, different measurements serve different purposes. Some measurements may be nonobvious but useful for tracking the success of a project and community relative to internal goals. Others may be better suited for a press release or other external consumption. For example, as a community manager, you may really care about the number of meetups, mentoring sessions, and virtual briefings your community has held over the past three months. But it's the number of contributions and contributors that are more likely to grab the headlines. You probably care about those too. But maybe not as much depending upon your current priorities.

Still other measurements may relate to the goals of any sponsoring organizations. The measurements most relevant for projects tied to commercial products are likely to be different from pure community efforts.

Because communities differ and goals differ, it's not possible to just compile a metrics checklist, but here are some ideas to think about. As Red Hat's James Falkner notes: "Each community has its own mission, goals, and central thoughts around which people collaborate, and the measurements you take must take those into account, without wasting yours or anyone else's time." The following are some common threads and themes with respect to measuring community and project health.

Understand Softer Aspects of Community

Conducting surveys and other studies can be time consuming, especially if they're rigorous enough to yield better than anecdotal data. It also requires rigor to construct studies so that they can be used to track changes over time. In other words, it's a lot easier to measure quantitative contributor activity than it is to suss out if the community members are happier about their participation today than they were a year ago. However, given the importance of culture to the health of a community, measuring it in a systematic way can be a worthwhile exercise.

Breadth of community, including how many are unaffiliated with commercial entities, is important for many projects. The greater the breadth, the greater the potential leverage of the open source development process. It can also be instructive to see how companies and individuals are contributing. We saw earlier how Ansible contributions benefit from the modularity of the software. Projects can be explicitly designed to better accommodate casual contributors.

Are new contributors able to have an impact or are they ignored? How long does it take for code contributions to get committed? How long does it take for a reported bug to be fixed or otherwise responded to? If they asked a question in a forum, did anyone answer them? In other words, are you letting contributors contribute?

Advancement within the project is also an important metric. Mikeal Rogers of the Node.js community tells how: “The shift that we made was to create a support system and an education system to take a user and turn them into a contributor, first at a very low level and educate them to bring them into the committer pool and eventually into the maintainer pool. The end result of this is that we have a wide range of skillsets. Rather than trying to attract phenomenal developers, we’re creating new phenomenal developers.”

Whatever metrics you choose, don’t forget why you made them metrics in the first place. I find a great question to ask is, “What am I going to do with this number?” If the answer is to just put it in a report or in a press release, that’s not a great answer. Metrics should be measurements that tell you either that you’re on the right path or that you need to take specific actions to course correct.

For this reason, Stormy Peters, who handles community leads at Red Hat, argues for keeping it simple. She writes that “It’s much better to have one or two key metrics than to worry about all the possible metrics. You can capture all the possible metrics, but as a project, you should focus on moving one. It’s also better to have a simple metric that correlates directly to something in the real world than a metric that is a complicated formula or ration between multiple things. As project members make decisions, you want them to be able to intuitively feel whether or not it will affect the project’s key metric in the right direction.”¹⁸

Back to the Bazaar

With healthy and well-governed communities, open source software can be an extremely effective software development model. We see examples everywhere in the software landscape today. The degree to which open source can reduce the friction of companies and individuals to collaborate took a long time to fully appreciate. Open source software

¹⁸<https://medium.com/open-source-communities/3-important-things-to-consider-when-measuring-your-success-50e21ad82858>

may have often begun life as a lower cost alternative to proprietary products. But today, whole categories of software default to open source in whole or in part.

That said, we can also look around and observe that, important as open source is, it's not the entire software universe. It's reasonable to ask why and where the open source development model hasn't displaced the alternatives.

Part of the reason relates to the business models that companies can successfully build in conjunction with open source software and open source development practices. I'll dive deeper on that topic later in this book. However, for now, suffice it to say that many companies built themselves around proprietary software development processes. Others use open source software to greater or lesser degrees but use it together with proprietary code and services. This doesn't mean an open source development model couldn't be effective in these cases, but rather that it's not a fit with how they want to run their businesses.

But what of the open source development process itself? It comes back to the hubbub of the bazaar, which no software program developed in an open community can escape entirely. A few statements about the nature of that bazaar broadly hold true.

It's a Bit of a Free-for-All

Open source tends to be about lack of restrictions, lack of forced approaches, and lack of guard rails. Individual projects aren't quite that freewheeling but upstream projects are often still likely to include features that are early-stage, experimental, or sometimes just broken. Commercial products mitigate this confusing picture to a certain degree by, paradoxically, constraining options and opinions.

Of course, the old proprietary world had plenty of differing opinions to choose from as well—and once you bought into one, you couldn't easily change your mind. Nonetheless, we can generalize that the open source landscape can be chaotic to navigate in the absence of a guide.

Open Source Is Duplicative

How many stalls filled with similar bangles and beads do you really need anyway? Shouldn't someone just herd them all into one organized marketplace?

Sounds like open source. Whether it's the many graphical user interfaces on desktop Linux systems, the almost overwhelming number of overlapping projects in the cloud-native computing space, or the almost countless number of Linux distributions, the

apparent inefficiency can seem staggering. At best, it's Darwinian with a smaller number of successful projects gaining community and momentum over time—as Kubernetes has largely done in the container orchestration space.

This is one of the ways in which innovation happens but it's a messy process.

Community Makes It Work

Ultimately, effective use of the open source development model requires that a community be able to coalesce and be willing and able to contribute to a project at the level needed to make it successful.

If we look across the open source software landscape at many of the successful projects, we can start to discern certain patterns. Linux, OpenStack, Kubernetes, the Apache web server, Hyperledger. They're all infrastructure software projects that have broad horizontal applicability.

What you don't see much of is payroll, financial accounting, or software that's very specific to a particular profession or industry. There probably aren't a lot of people who want to casually contribute to dental records management in their spare time or who have either the inclination or the knowledge to update tax rules in an application every year.

Even when there are open source versions for these types of applications, the open source code tends to be primarily written by one company and exist in the market alongside proprietary “Pro” offerings with more features or hosted services.

Furthermore, popularity doesn't equate to funding. Even beyond infrastructure, which we've discussed, there are many popular projects in programming languages, creative apps, and elsewhere that lack significant enterprise-level funding. Funding matters because the popular image of open source software being developed mostly by hobbyists on nights and weekends doesn't apply to just about any major open source project. And large communities also need support through sponsorships and other funding just to keep the lights on for basic community maintenance.

Open source communities can be an extremely powerful model to develop software. Creating and growing a community has to be deliberate work that's matched with the nature of and goals for the project. And it may still not work if another project gains the mindshare or if the project just isn't suited to community-based development. You can't put out a sign that says, “Work on my stuff for free” and expect it to just happen as if by magic. You have to understand the open source development model and approach it deliberately.

Why the Development Model Matters

The open source development model wasn't originally the featured headline of open source. But user freedoms are amplified when there are effective communities developing the software. Especially as projects grow, that requires appropriate governance structures, community management, and having a meal or drinks together now and then.

CHAPTER 4

Open Source's Connection to the Past

Open source grew out of specific aspects of the computer industry, but it's also tied to how companies have collaborated—and innovation has taken place—historically. Information, intellectual property, has long been informally shared even absent a formal, or sanctioned, process. Researchers have also studied how communication takes place and how people are motivated long before there was such a thing as open source software. But open source both benefits from such research and provides fodder for expanding on these topics going forward.

The Machine That Drives Open Source

As we learned in the last chapter, clearly there's something new, different, and powerful in the open source development model and the way that it can be used by successful businesses. That makes understanding how and why open source works a matter of more than academic interest even if it's often academics who formally study how open source software development works as a process.

Furthermore, there's a suspicion—more than a suspicion really—that open source has insights to offer that go beyond software development. How best to innovate? How do people communicate and self-organize? What intrinsically motivates individuals to take on certain tasks? What should we be measuring in organizations?

These are all reasons to peer within open source software projects and communities, ponder possible learnings, and consider what they may tell us about other aspects of interaction and organization within companies, between companies, and among individuals in both their professional and personal spheres. Open source is, after all, a product of a highly interconnected world. It's reasonable to assume that the forces ushering in open source also have effects elsewhere.

Innovation

Innovation has been the subject of academic study for a long time. After all, technology has been dramatically changing how humans live ever since there have been humans. Even the Industrial Revolution was just a dramatic acceleration of a long-standing trend.

Innovation Through Collective Invention

One early paper that specifically focused on collective invention came from Robert C. Allen of the University of British Columbia in 1983.¹

Allen was studying the 19th-century iron and steel industry and seeking to better understand the manner in which inventions took place. It was a milieu with little or no government funding and minimal academic research; even the firms themselves didn't pursue research and development in any formal way. With the exception of a few well-known inventors like Bessemer, the process seems to have been very diffuse.

In the course of his research, Allen concluded that a great deal of information about furnace designs was widely shared. He based this, in part, on the designs that companies made publicly available.

But it also stemmed from the nature of the inventions themselves. Most changes were incremental and even experimental in a field that generally lacked theories about how to build a better blast furnace from first principles. (A characteristic of much open source software as well.)

Allen speculated that output and profits could be greater if the "existing regime of trade secrets was replaced by a new regime of free information exchange." He also thought that firms might release information because "so many people would know the relevant information that it would have been costly to keep it secret." Secrecy can introduce frictions and transaction costs.

In conclusion, he writes: "An essential feature of collective invention was the release of technical information to actual and potential competitors. It was this behavior which allowed cumulative advance. All of the factors that account for this behaviour apply to the other institutions as well. Hence, one would expect to observe the wilful dissemination of technical knowledge under a variety of circumstances. And, indeed, even a casual acquaintance with recent engineering literature indicates that such behaviour is rampant

¹Allen, R. C. (1983). "Collective Invention." *Journal of Economic Behavior and Organization*. Volume 4. Issue 1. March 1983 Pages 1-24.

today. To the degree that economists have considered this behaviour at all, it has been regarded as an undesired 'leakage' that reduces the incentives to invent. That firms desire such behaviour and that it increases the rate of invention by allowing cumulative advance are possibilities not yet explored. They should be."

The Economics of Open

Eric von Hippel of MIT's Sloan School of Management has made something of a career out of studying innovation going back to the 1970s. But "spurred on by [Allen's] work and the evident success of open source software, quite a few of us then began trying to understand the economic benefits of open vs. proprietary information."² For example, in 1987, he wrote that "'informal' know-how trading is the extensive exchange of proprietary know-how by informal networks of process engineers in rival (and non-rival) firms. I have observed such know-how trading networks to be very active in the US steel minimill industry and elsewhere, and they appear to represent a novel form of cooperative R&D."³

Von Hippel and ETH Zurich's Georg von Krogh argue that "society has a vital interest in encouraging and rewarding innovation."⁴ They write about how "open source software development is an exemplar of a compound model of innovation that contains elements of both the private investment and the collective action models."

The private investment model is how products were traditionally brought to market and commercial software written. Innovation is supported by private investment with the profits going to the entity making the investments. With this model, any "spillover" of proprietary knowledge is considered to reduce profits. Hence, the existence of trade secrets, patents, and other ways to reduce the ability of others to make money off innovations that you spent money creating.

By contrast, the collective action model applies to public goods, which have two characteristics. First, if even one user consumes them, every other user also has access to them. Second, the act of consuming the good doesn't mean there's less of it for everyone else. In economist speak, a public good is defined by its non-excludability

²<https://evhippel.mit.edu/papers/section-3/>

³von Hippel, Eric. "Cooperation Between Rivals: Informal Know-how Trading." *Research Policy* 16, no. 6 (December 1987).

⁴von Hippel, Eric, and Georg von Krogh. "Open Source Software and the 'Private-collective' Innovation Model: Issues for Organization Science." *Organization Science* 14, no. 2 (March 2003).

and non-rivalry. The collective action model operates in science and elsewhere. If a researcher publishes a paper proposing a novel use for a drug, that knowledge is now available to everyone and one person making use of that knowledge doesn't mean that there's now less knowledge for everyone else. It's sometimes described as an example of something you do in the event of a market failure in which problems like the tragedy of the commons prevent market mechanisms from functioning properly.

They then add that "In the case of open source software development projects, we see an interesting compound of the private investment and collective action models of innovation. We term this compound the 'private-collective' innovation model. In this model, participants in open source software projects use their own resources to privately invest in creating novel software code. In principle, these innovators could then claim proprietary rights over their code, but instead they choose to freely reveal it as a public good. Clearly, the net result of this behavior appears to offer society the best of both worlds—new knowledge is created by private funding and then offered freely to all."

The Advantages of Collaborative Innovation

Von Hippel and von Krogh wrote those words in 2003, still relatively early days for commercial open source software and even earlier days for open source driving large amounts of innovation as opposed to being primarily a less expensive alternative to proprietary software. However, they already recognized that something different was going on with the relationship of open source software to innovation that wasn't captured by traditional economic models.

In 2017, von Hippel expanded on his innovation work in *Free Innovation* (MIT Press). This book specifically focuses on innovation created by individuals in the absence of commercial transactions. However, many of the same principles apply to organizations participating in open source projects.

For example, he writes that "a collaborative innovation project offers two major advantages over innovation projects carried out by individual free innovators. The first major benefit from a participant's perspective has to do with output value obtained: Each individual participant incurs the design cost of doing a fraction of the project work but, if intending to use it, obtains the value of the entire design, including additions and improvements generated by others . . . A second major advantage of collaborative projects over single innovator projects is that collaborative projects greatly expand the range of innovation opportunities that are viable for free innovators. This is because overall project costs are no longer limited to a level of design costs that are viable for a single individual."

He adds that “protective measures would shrink the pool of potential contributors, and so shrink the overall scale of the project. The network properties of the collaborative innovation model (the fact that the value to everyone increases as the total number of contributors increases) mean that this reduction in the contributor pool would reduce the value of the project to the contributors who remain as well as to free riders.” This reinforces the idea that it’s usually better to focus on taking full advantage of the open source development model rather than worrying about free-ridership.

How Knowledge Gets Shared

Other research has focused on the role of networks and ties in innovation, including the sharing of knowledge.

For example, in 2008, Annapoornima Subramanian and Pek-Hooi Soh of the National University of Singapore did an analysis of open source projects through the lens of knowledge integration theory. They looked at the efficiency of integration, the extent to which the projects accessed and utilized the specialized knowledge held by individual members. They considered flexibility of integration, the extent to which a project can access additional knowledge and reconfigure existing knowledge.

This research drew conclusions similar to other studies that emphasize the importance of communication in open source projects. The authors write that “In an open knowledge sharing environment such as OSS [open source software], effectiveness of the project greatly depends on the extent to which the project can integrate and build upon knowledge that is available in the public domain.” They conclude that this phenomenon, also known as open innovation, is essential for technologies such as software, where the knowledge required for building the innovation is widely available.⁵

Collaboration and Communication

Inherent in cooperative ventures is the need to collaborate and communicate. Open source software projects—as well as other open projects like Wikipedia—have been a rich data trove for researchers in this area because so much communicating takes place on public mailing lists, code repositories, and chat systems like IRC. Open source

⁵“Knowledge Integration and Effectiveness of Open Source Development Projects.” *IIMB Management Review*, June 2008.

projects are also interesting, if complicated as a result, because they can be a mix of people contributing on their own time and people contributing as part of their day job, or even a blend of the two.

All this makes open source software development a useful study point for distributed teams generally. What's the effect of geographic distribution? Does being spread across many different time zones matter? What self-organizing structures appear? How does organization of a project team affect the resulting software?

The Limits of Communication

Take Conway's law. It's an adage named after computer programmer Melvin Conway, who argued in a 1968 *Datamation* article that "any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure."⁶ A common shorthand restatement is that "If you have four groups working on a compiler, you'll get a four-pass compiler."

Conway's observation was mostly empirical. However, it rings true for many practitioners. A relatively recent example comes from an Amazon offsite meeting at which managers suggested that employees should be communicating with each other better—a perennial complaint that's pretty much par for the course at just about any large company. To their surprise, founder and CEO Jeff Bezos stood up and announced, "No, communication is terrible!"⁷ Bezos' idea was that communication works better within small "two-pizza teams" (teams no bigger than two pizzas can feed) and that communications among teams can be minimized by requiring that interactions between software written by each team should only take place through well-documented public interfaces.

This is one of the ideas behind microservices, or service-oriented architectures more broadly. So long as you stick to the contract—that is, the documented interface—any team is theoretically able to make whatever changes they want to their code, including rewriting it in a new language or changing algorithms, so long as they don't make changes to the public application programming interface (API).

⁶http://www.melconway.com/Home/Committees_Paper.html

⁷<http://blog.idonethis.com/two-pizza-team/>

In one study of the relationship between software and teams writing the software,⁸ researchers found that as teams increase in size, forecasts regarding how long it would take to develop some piece of software tend to get more optimistic. (This bias shone through whether the forecast came from the team itself or someone outside the team.) Furthermore, they found that even though large teams thought they would be faster than smaller ones, they could actually be slower. This echoes Fred Brooks's observation in *The Mythical Man Month* (Addison-Wesley Professional, 1975), an account of the design of the IBM System/360 family of computers, that "adding manpower to a late software project makes it later."

The problem, as organizational psychologist and expert on team dynamics J. Richard Hackman has pointed out in *The Psychology of Leadership: New Perspectives and Research* (Psychology Press, 2004), is that the number of links between people increases by on the order of the square of the number of people. And it's the number of links that correspond to the communication difficulty and overhead.⁹

How Communication Affects Software Structure

Other research from MIT and Harvard Business School¹⁰ compared the structure of proprietary software relative to open source counterparts developed by more distributed and loosely coupled teams. They concluded that the product developed by the loosely coupled organization was significantly more modular—that is, design changes in one component don't affect other components—than the product from the tightly coupled organization.

This is in keeping with the observations of community managers and others that it's easier to create communities around open source software that's modular. When software is monolithic and the team is distributed, the organizational structure is effectively fighting against the structure of the software.

There has also been research into social group size, which is potentially interesting for understanding how large open source communities—or at least their subgroups—

⁸http://www.opim.wharton.upenn.edu/~kmilkman/2012_OBHPb.pdf

⁹To be precise, it's $n(n-1)/2$. Ten people have 45 links. One hundred people have 4,950 and so forth.

¹⁰https://www.hbs.edu/faculty/Publication%20Files/08-039_1861e507-1dc1-4602-85b8-90d71559d85b.pdf

can grow while still being effective. One frequently quoted number here is Dunbar's¹¹ number—popularized in Malcolm Gladwell's *The Tipping Point: How Little Things Can Make a Big Difference* (Little, Brown, 2000)—which is based on the idea that “there is a cognitive limit to the number of individuals with whom any one person can maintain stable relationships, that this limit is a direct function of relative neocortex size, and that this in turn limits group size.”

That number is 150 people and is widely interpreted as the maximum size for cohesive social groups. However, as noted by Christopher Allen, that's misleading—or at least incomplete.¹² Dunbar's number applies to groups that have a strong incentive to stay together, which may not be the case in many open source communities. Lacking strong incentives, the number may be smaller. Allen cites some numbers from online gaming communities that suggest 50 to 60 active participants in a group may be a more realistic upper number.

Modularity Is Generally Better

Concepts like modularity and a desire to constrain the amount of communication that needs to take place comes up time and time again in open source project data.

In 2008, Richard N. Langlois and Giampaolo Garzarelli argued that “open-source collaboration ultimately relies on the institutions of modularity.”¹³ They observed that, around the same time that Brooks was coining the “mythical man-month” phrase, David Parnas (1972) and other researchers were looking at software systems in a different light. Solving the problem of interdependency, these researchers argued, is not a matter of maximizing communication among the parts but rather of minimizing communication. Just as Bezos told his managers.

Systems taking this approach are built from parts that do not need to communicate extensively with one another. In fact, they are forbidden from doing so by design. They're also, at least in principle, reusable. Parnas argued that “system details that are likely to

¹¹<https://www.cambridge.org/core/journals/behavioral-and-brain-sciences/article/coevolution-of-neocortical-size-group-size-and-language-in-humans/4290FF4D7362511136B9A15A96E74FEF>

¹²http://www.lifewithalacrity.com/2004/03/the_dunbar_numb.html

¹³Langlois, Richard N., and Giampaolo Garzarelli (2008). “Of Hackers and Hairdressers: Modularity and the Organizational Economics of Open-source Collaboration, Industry and Innovation.” *Industry and Innovation*, Volume 15, 2008 - Issue 2: Online Communities and Open Innovation.

change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change.”

Langlois and Garzarelli conclude that modularity is generally superior to monoliths (or integrality as they call it) across most dimensions in both software design and organizational structure. It's only in the setup costs that modularity costs more. This is consistent with what practitioners today observe with microservices and other modular software design patterns. There's an initial cost to carefully mapping out the service boundaries and determining how components will communicate with each other. The effort may not be justified for a small project and team. But modularity tends to win out as projects and teams grow.

How Contributors Interact in Open Source

Other research has looked at the interaction patterns in specific projects.

An analysis of Apache Software Foundation projects showed “that a small group of contributors is responsible for the majority of commits.”¹⁴ This is a common pattern in many open source projects and reinforces the importance of supporting casual contributors.

Jorge Colazo of Trinity University found that “. . . more successful teams in terms of both quality and productivity, seem to rely more on ‘linchpin developers’ that become more active as the team’s TD [temporal distribution] grows and serve as ‘bridges’ between time zones”¹⁵ in their study of 230 open source software projects. This sounds a lot like the role of maintainers who often have key responsibilities in large projects.

A 2012 study from Microsoft Research on the Eclipse IDE and the Firefox web browser code bases found that, while the developers of large modules could be geographically distributed, there was a tendency for smaller modules to be developed by contributors at a single site.¹⁶

The relative importance of location may vary by project though. Studying collaboration on the Linux kernel mailing list and interviewing developers, Dawn Foster

¹⁴Chelkowski, Tadeusz, Peter Gloor, and Dariusz Jemielniak, *Inequalities in Open Source Software Development: Analysis of Contributor’s Commits in Apache Software Foundation Projects*

¹⁵Colazo, J. (2014). “Structural Changes Associated with the Temporal Dispersion of Teams: Evidence from Open Source Software Projects.” 47th Hawaii International Conference on System Science.

¹⁶Christian Bird and Nachiappan Nagappan, “Who? Where? What? Examining Distributed Development in Two Large Open Source Projects.” Published in 2012 9th IEEE Working Conference on Mining Software Repositories (MSR).

concludes that “Location doesn’t matter.”¹⁷ Among the things developers told her were “The Linux community doesn’t care where you’re located, ever. You can be on the moon as long as you have a good Internet connection.” And “Similar time zones can be helpful because I can get a reply immediately. But it is not super important.”

What was important? Relationships, including the ability to get together face to face. One developer told her, “At conferences you can really sit down with a beer, hash things out, and come to a consensus. The Linux Kernel Summit is massively useful for that kind of thing.”

Foster’s quantitative analysis bear out the interview results. Looking at the USB Linux kernel mailing list specifically, she found that social networks (previous interactions) and working for the same organization affected the level of interactions, but time zone did not.

Participation

How do you encourage participation in open source projects, and what are the motivations for developers and others to do so?

How Participants Get Started

In 2014, researchers from the University of Helsinki and Stanford took a look at onboarding in open source projects.¹⁸ They began by observing that “the general problem of introducing new people into an existing organization is especially pronounced in distributed settings with virtual teams—small, often temporary groups of knowledge workers, separated by geographical, temporal, and cultural distances that add to the communication challenges involved in getting them to work effectively.”

As we’ve seen in other work, geographic separation isn’t necessarily a big issue in well-established projects among participants who already know each other. However, gaining initial trust can require different techniques than retaining participants and encouraging their ongoing involvement.

¹⁷<https://www.slideshare.net/geekygirldawn/collaboration-in-linux-kernel-mailing-lists-81009604>

¹⁸Fagerholm, Fabian, Alejandro Sanchez Guinea, Jürgen Münch, and Jay Borenstein, “Onboarding in Open Source Projects,” IEEE Software, November/December 2014.

This study focused specifically on this very early stage of integrating newcomers or “climbing the learning curve in virtual teams” as the authors put it. This is important because, as we’ve seen, people come and go all the time in healthy communities so efficiently bringing new members into the fold needs to happen continuously unless a community is to wither. Their study point was somewhat artificial—a short-lived collaboration project between Facebook and a number of universities to get students involved with open source. Nonetheless, it’s interesting to see data that backs up the more anecdotal experience of communities like Kubernetes with respect to mentoring.

Onboarding and Mentoring

In this case, the onboarding process consisted of two things: a co-located hackathon and mentoring; the mentoring subsequently took place through the usual open source project communications channels such as email and IRC chat. Although the study focused on the mentoring, the hackathon is also notable; virtual collaboration tends to work better when “in real life” get togethers are part of the mix. Mentors were tasked with recommending and detailing tasks, explaining the software architecture, and assisting in technical development details.

The students joined the project of their choice using the usual processes already in place for the project. Mentors would guide them to small tasks of interest suitable for a newcomer, eventually letting them take on tasks of greater complexity when they felt up to it.

The researchers then measured the activity of the mentored students as well as that of a random group of other equivalently inexperienced project participants. Activity was defined as the total number of commits, pull requests, and interactions by each developer as a proxy for how involved they were with the project.

The results showed that mentoring can have a significant positive impact. By the end of 16 weeks, the mentored developers had about twice the level of activity of the unmentored ones. It’s worth observing that it took about three weeks for there to be any real difference. This suggests that mentoring isn’t a quick hit, one-off task. It’s an ongoing process.

The study authors acknowledge that this means the productivity of the developers doing the mentoring is going to be reduced; the amount varied by project but it ranged from about 20 to 35 percent reduction. But they concluded that “While the performance drop can be significant, it can be justified by the fact that it’s limited to the onboarding

period, and by the potential impact on increased productivity and increased innovation that could be gained from new project members. In practice, this opportunity cost must be evaluated on a case-by-case basis. In many situations, the cost of mentoring is overshadowed by the potential benefits of gaining new members, even if they're temporary."

Motivation

When we talk about motivations, one common way to do so is in terms of incentive theory. This theory began to emerge during the 1940s and 1950s, building on the earlier drive-reduction theories established by psychologists such as Clark Hull and Kenneth Spence. Incentive theory was originally based on the idea that motivation is largely fueled by the prospect of an external reward or incentive.

The Types of Motivators

However, more recent research beginning in the 1970s has also focused on intrinsic motivations, which do not require an apparent reward other than the activity itself. Self-Determination Theory, developed by Edward Deci and Richard Ryan, would later evolve from studies comparing intrinsic and extrinsic motives, and from a growing understanding of the dominant role intrinsic motivations can play in behavior.

Money is a classic extrinsic motivator. So is winning an award, getting a grade, or obtaining a certification that is likely to lead to a better job. Intrinsic motivations usually come from simply doing activities that you enjoy in some fashion. You play in a softball league after work because you like playing softball. Others have proposed a further distinction between this enjoyment-based intrinsic motivation and obligation/community-based intrinsic motivation, which is more about adherence to social or community norms. Maybe you don't really like having the relatives over for Thanksgiving, but you do it anyway because you know you should.

Today's psychology literature also includes the idea of internalized extrinsic motivations. These are extrinsic motivations such as gaining skills to enhance career opportunities—but they've been internalized so that the motivation comes from within rather than coming directly as a result of a carrot dangled by someone else.

Research into Open Source Motivations

In 2012, four researchers at ETH Zurich (G. V. Krogh, S. Haefliger, S. Spaeth, and M. W. Wallin) surveyed the prior 10 years of study into open source software contribution based on the top 200 articles as ranked by Google Scholar.¹⁹ Among their results, they were able to group study findings into the above three categories: intrinsic motivation, internalized extrinsic motivation, and extrinsic motivation—as well as finding some common subgroupings within those broader categories.

Intrinsic Motivators

Among intrinsic motivators, ideology and altruism often seem closely related. Free software was primarily an ideological statement at first, even if user control also had an important practical side; several researchers have found support for ideological motives in developer surveys.

Altruism can also be a developer motive. The research results are mixed. One paper identified the “desire to give a present to the programmer community” as a crucial pattern in open source software. But other studies have qualified the importance of altruism as a motive. For example, a 2002 study by Hars and Ou said that, while 24 percent of student and hobby programmers said altruism was their biggest motivation, only 8 percent of the programmers that paid for open source development did so.

Extrinsic motivations can displace intrinsic ones. Or, absent a sufficient paycheck, many will decline to participate at all by reason of preference or necessity. Altruism doesn't pay the rent. Other work found that altruism could be a motivator but only among developers who were otherwise satisfied.

Kinship amity is another potential intrinsic motivator. It's related to the concept of gift economies but is specific to family (kin) groups that don't expect a calculated quid pro quo. (In other words, family members do things for each other but they don't typically keep a running tally, at least a systematic one, of who hasn't been pulling their weight recently.) It's also different from altruism in that it is restricted to the group to which one belongs, such as an open source community. Here again the research is mixed. Studies have generally found a positive relationship between identified kinship

¹⁹Krogh, G. V., S. Haefliger, S. Spaeth, and M. W. Wallin (2012). Carrots and Rainbows: Motivation and Social Proactive in Open Source Software Development. *MIS Quarterly*, Vol. 36, No. 2 (June 2012), pp. 649-676.

amity and various measurements of effort, such as number of hours worked per week—though some have found weaker relationships than others.

There's also the motivational power of fun and enjoyment, a classic intrinsic motivator. This should come as no surprise to anyone who hangs around open source developers. Almost all of them *like* working on open source projects. One large study by Luthiger and Jungwirth in 2007 of 1,330 open source developers determined that fun accounted for 28 percent of the effort (in terms of number of hours) dedicated to projects. One implication of this research is that activities that developers perceive to be more of a drudgery, such as technical support often is, may require alternative forms of motivation.

Internalized Extrinsic Motivators

Much of the research into reputation as an internalized extrinsic motivator has focused specifically on peer recognition. Peer reputation is usually targeted at community insiders and potential employers as a way of signaling talent. The suggestion that reputation could be an important motivator in contributing to open source goes back at least as far as a 1998 Eric Raymond essay, “Homesteading the Noosphere.” However, since then, a variety of surveys have supported the idea that peer reputation is a driver for participation. A 2005 paper by Lakhani and Wolf found peer reputation to be the fourth biggest determinant of invested effort.

Learning is frequently mentioned as a big benefit to participating in open source projects. Studies more or less universally bear this out. There's debate over what learning means in this context though. Is it learning for learning's sake? Is it learning for skills development? The authors of this ETH Zurich survey noted that learning was almost ubiquitous as a motivator throughout the literature that they examined, but what “learning” actually signified was often unclear.

A final motivator in this category is what researchers call “own-use value” but is more recognizably described as something like “scratch your own itch.” Develop something that you want for yourself and create something for others in the process. That something is ultimately an external reward to yourself, but no one is forcing you to do it.

It's unsurprising that researchers would find this was a good motivator. Certainly the folk wisdom is that *many* developers get into open source to scratch an itch by developing something they themselves need—such as when Linus Torvalds wrote Git because Linux needed an appropriate distributed version control system.

However, other research warns of certain downsides to this motivation, if taken to an extreme. For example, one 2006 study reported that developers scratching their own itch worked “eclectically”; they would fix bugs that annoyed them and then quit until the next time, making fewer contributions over the long haul as a result.

Extrinsic Motivators

With extrinsic motivation, we come to the sorts of motivators that everyone is familiar with.

There’s money of course. But are open source contributors paid? Yes, in many cases, outside of personal or hobby projects or occasional casual nonprofessional contributors. A significant portion of the literature studied in this survey dated to the early-mid 2000s and thus probably reflects a somewhat earlier and less commercial era of open source than we are in today. At a minimum, there were fewer large projects associated with successful commercially supported products than is the case today. Even so, most of the open source projects studied had a significant number of contributors who were paid by their companies to work on open source.

Career goes hand in hand with pay. But is open source software any different from proprietary software development in this regard? There’s some evidence that it is. Lerner and Tirole first suggested in 2002 that “individual developers would be motivated by career concerns when developing open source software. By publishing software that was free for all to inspect, they could signal their talent to potential employers and thus increase their value in the labor market.” There’s some support from in the literature in favor of this thesis.

More recently, there’s been significant empirical evidence to support the argument that there are career advantages to developing code that’s in the open and therefore available for others to look at and work with. It seems to have become almost an expectation in some segments of the industry that job applicants have public GitHub code repositories that are effectively part of their resume. It’s reasonable to ask whether this trend has gone too far. After all, many highly qualified developers work on proprietary code. But it’s clear that, fair or not, at least some career prospects are tied to being an open source developer.

Measurement

I touched on the principles of metrics earlier, but it's worthwhile to close out the topic of research with a deeper dive into appropriate measurements for measuring both the maturity of a project's software development process and its broader overall health.

Why Measure?

But why do we take measurements anyway? Is it to comfort us that everything is humming along without a glitch? Is it to warn us that something is amiss? Is it to provide evidence in the event of some unanticipated future meltdown?

Measurement can do all those things but, fundamentally, it's the means by which we inform intelligent decisions and actions. Writing in 1973, Peter Drucker (who once appeared on a *Business Week* magazine cover as "The Man Who Invented Management") argued that measurement is the basic element in the work of a manager. And, furthermore, few factors are as important to the success of an organization as yardsticks based on measurements.

Measurements Affect Behaviors

At the same time, it's important not to think of measurement as a dispassionate mechanistic system that sits outside the people and process being measured. As Robert Austin writes in *Measuring and Managing Performance in Organizations* (Dorset House, 1996): "Organizational subsystems are composed of people, most of whom maintain among their goals the desire to look good in the eyes of those responsible for evaluating and allocating rewards to the subordinate subsystems. The desire to be viewed favorably provides an incentive for people being measured to tailor, supplement, repackage, and censor information that flows upward." If you're responsible for a measurement that makes you look bad, others shouldn't be surprised if it goes missing or you reinterpret it in a more favorable light.

Much of the more formal work intended to formalize and standardize metrics—including that based on traditional measures of software development activity—has tended to sidestep such cautions.

For example, IEEE Standard 1061 proposes direct metrics as a way to quantify attributes so that they can be compared to a target value. The idea is that direct measures, such as defects found during testing, are independent measures that aren't affected by other variables.

The Limits of Direct Measures

However, matters are rarely that simple. Writing for the 10th Annual Software Metrics Symposium in 2004, Cem Kaner and Walter Bond noted that “As soon as we include humans in the context of anything that we measure—and most software is designed by, constructed by, tested by, managed by, and/or used by humans—a wide array of system-affecting variables come with them. We ignore those variables at our peril. But if we take those variables into account, the values of our seemingly simple, ‘direct’ measurements turn out to be values of a challenging, multidimensional function.”²⁰

Part of the problem is that there are, in fact, relatively few measurements that are both direct and useful. Quality of code is probably an unambiguously good thing. But that's mostly because we're effectively defining quality as an amorphous good as opposed to something that we can directly measure. Instead, we measure attributes like test coverage or bugs found. But what if we're doing a great job of finding minor surface problems that don't have much of an impact while missing deep-rooted flaws that could be showstoppers? Even under the best of circumstances, the proxy or surrogate measures that we're forced to use are ambiguous and indirect measures of the attribute we actually care about.

Measurements at Cross-Purposes

Another common problem arises from how a given measurement can be seen as either fulfilling a motivational or an informational purpose.

Motivational measurements are explicitly intended to affect the people being measured. If you don't do well enough on the certification exam, you won't pass, and you won't be able to apply for the better-paying job that requires the certification. Measurements of this type may also provide some guidance as to which skills need further development or where your aptitude may be highest. They may also be useful for

²⁰Kaner and Bond, “Software Engineering Metrics: What Do They Measure and How Do We Know?”

evaluating the effectiveness of training and hiring programs in the aggregate. However, for the most part, the primary purpose of motivational measurement is to provide rewards and change behavior.

By contrast, Austin describes informational measurements as being “valued primarily for the logistical, status, and research information they convey, which provides insights and allows better short-term management and long-term improvement of organizational processes.” As a result, you ideally want people to behave as if these measurement systems weren’t even in place.

The software development version of the Heisenberg Uncertainty Principle means that this is a hard constraint to keep in place. You measure something and you tend to change it. In practice, it’s hard to view processes that the numbers say are broken without reflecting on the role that individual human failures may have played. (And the individual humans know this.) It may in fact be true that attempting to force compatibility between motivational and informational metrics can cause “dysfunction,” to use Austin’s word. But it’s a common reality of measurement systems that we should take into account as much as possible.

DevOps and Metrics

Much of the more recent research into software measurements is more empirical and revolves around a more iterative DevOps model for developing software.

A traditional software development lens tends to view the release of a software version as something akin to shipping any other product. True, software makes it possible to ship bug fixes and updates over time. But there’s still a finality to pressing a gold master image at the end of a release cycle and shipping it out into the world. One stage ends and another begins. The software passes from the world of development to the world of operations. The measurements reflect this. Software development and quality metrics for a waterfall world focus on aspects like the percentage of code complete and the decline of open bug reports over time.

By contrast, DevOps measurements are more about the ongoing health and effectiveness of a software process throughout its entire life cycle. *The State of DevOps*, an annual report most recently presented by Puppet Labs and DevOps Research & Assessment (DORA) in 2017, offers an insight into relevant metrics that can be tracked over time. The report notes that “We measure IT performance along two main dimensions: throughput of code and stability of systems. Throughput is measured by

how frequently a team is able to deploy code and how fast it can move from committing code to deploying it. Stability is measured by how quickly the system can recover from downtime and how many changes succeed, versus how many fail.”

Throughput Measures

With respect to throughput, *The State of DevOps* looks at deployment frequency and change lead time. To measure stability, it considers mean time to recover and change failure rate.

Deployment frequency is one of the most striking metrics compared to traditional software release methodologies. Higher-performing organizations “reported that they are routinely deploying on demand, performing multiple deployments per day.” These kind of numbers—to say nothing of outliers like Amazon and Netflix, which release thousands of times per day—are still outliers in many organizations and projects. But other research such as a 2015 IDC DevOps Thought Leadership Survey confirms that the “majority of DevOps users expect to significantly increase the rate of code releases in the next two years”—from what were often annual releases to monthly and even more frequent ones.²¹

In general, DevOps measurements of this type reveal a stark gap between high and low performing organizations. For example, *The State of DevOps* says that “high performers reported that their lead time required to deploy changes into production (i.e., go from code committed to code deployed and running successfully in production) was less than one hour, whereas low performers required lead times between one week and one month.”

Stability Measures

Stability measurements like mean time to recover and change failure rate reveal a different mindset from traditional quality metrics. While you may (and probably will) also track conventional measures such as open bugs, the focus here is on reducing failure rate and minimizing the effects of failure rather than trying for a zero defect rate. “High performers reported a change failure rate between zero and 15 percent, while low performers reported change failure rates of 31–45 percent” according to the report.

²¹IDC DevOps Thought Leadership Survey, May 2015.

Many other metrics might matter throughout a DevOps process. Andi Mann of data analysis vendor Splunk suggests that, in addition to traditional software quality measures (tests passed or tests failed) and activity (number of commits and releases), you may also want to consider the impact of your software. For example, did the changes you made to a website increase the number of users signing up for your service?

The degree to which DevOps development and deployment patterns—and therefore the applicable measurements—apply to an open source development project will vary. However, the combination of modern development practices and the distributed and iterative nature of large open source projects suggests that the measurement approaches found to work for DevOps are a good place to start for open source projects as well.

Understanding Community Health

However, an open source project is much more than just a software development project. It's a community too. And monitoring the health of that community is at least as important as making sure that quality software is getting written and deployed.

The idea that there's value in systematically understanding how people feel about an organization they belong to or an activity they participate in isn't new. The first employee surveys, then commonly known as employee-attitude surveys, appear to have first surfaced in industrial companies in the 1920s. Over the next couple of decades or so, workplace motivation became a topic of academic study.

Morris S. Viteles was an influential researcher and writer in the field of industrial and organizational psychology and wrote the first comprehensive textbook about the field in 1932. Writing in *Motivation and Morale in Industry* in 1954 (W. W. Norton), he noted that “An interest in checking on morale conditions and finding ways to improve employee morale and relations within the company finds expression in reasons given for undertaking attitude surveys. Many companies expressed a desire to learn about the minor troublesome situations so that measures could be taken to prevent their growing into major ones. In general, statements made by the companies show clearly an expectation that the attitude survey would provide management with a measure of its own success or failure in personnel matters and, at the same time, locate unsatisfactory feelings and sources of irritation requiring remedial action.”

Shining More Light on Culture

Understanding how people fit into a software development project has been historically underappreciated and the metrics used fairly generic. However, DevOps's cultural emphasis has helped to shine a light on organizational measurements that can also be relevant to open source projects.

For example, a 2014 report from market researchers Gartner²² includes organizational effectiveness among its categories for “Data-driven DevOps” metrics. Some of the specific measures, such as retention, are fairly conventional and would apply to most positions within a company. However, Gartner also calls out mentoring, sharing, collaboration, and morale. These aren't metrics that most companies have focused on tracking historically. But they have clear relevance to open source projects and communities.

A new effort to define metrics for community health came out of the CHAOSS (Community Health Analytics OSS) Metrics Committee, a Linux Foundation project, in 2017.²³ Researchers from academia and practitioners from industry are getting together in CHAOSS to define a neutral, implementation-agnostic set of reference metrics to be used to describe communities in a common way.

The organization plans to initially focus on four open source project metrics: Project diversity and inclusion, project growth-maturity-decline, project risk, and value. These “complex metrics” are constructed from multiple “activity metrics.” Activity metrics include the usual dashboard numbers like new contributors and open issues but also factors that are less commonly tracked. For example, whether the path to leadership in the project has been published and how many rewards, shout-outs, recognition, and mentions there are in pull-requests or change logs.

Diversity and Inclusion

Diversity and inclusion considers a number of dimensions. There's individual diversity. What's the ratio of new contributors to maintainers? What are the demographics of contributors and leadership? Organizational diversity is about quantifying the degree to which a project is dominated by one or two companies. How many organizations are contributing and how many are represented by maintainers? How many contributions

²²“Data-Driven DevOps: Use Metrics to Help Guide Your Journey.” *Gartner*, May 2014.

²³<https://wiki.linuxfoundation.org/chaoss/metrics>

are coming from each organization? Inclusion seeks to identify how effective the community is in including and promoting members. When is the last time a maintainer was added?

Project Growth-Maturity-Decline

The goal of project growth-maturity-decline is to identify the level of maturity the project has reached and whether it is growing or declining. This involves measuring how effective the community is at addressing issues identified by community participants. For example, what's the mean or median age of open issues? It's also about the pace and effectiveness of merging new code. This is where traditional measures like the number of commits come in. However, it also brings in the difficulty and time associated with getting code contributions accepted. What is the duration of time for a maintainer to make a first response to a code merge request? Community size and growth are other basic measures that give a quick snapshot of the overall trajectory of a project.

Risk

Risk is a less fleshed-out metric at this point, but it's intended to be a set of factors that may pose a risk to an open source project depending upon the situation. For example, the percentage of commits made by paid developers over time suggest, at a minimum, a different character of project if the number is large versus when the number is small. Other risk factors include incomplete license or copyright information and high-profile security vulnerabilities.

Value

The final CHAOSS metric attempts to quantify value that developers and organizations capture through engaging in OSS communities. These include common measures like number of downloads and number of users. However, it can also incorporate an estimate for the market cost to develop the functionality of the project. This is admittedly a hard number to quantify. Historical software cost estimation models like Barry Boehm's Constructive Cost Model (COCOMO) in the 1970s have never been especially accurate. At least for some types of projects, another possibility is the V-index, which measures a project's first-order and second-order downstream dependencies. The more software

that incorporates a project through a dependency, the more value that project is creating.

These metrics will ultimately draw from a long list of activity measures that CHAOSS is developing. Some of these draw from automated and continuous data collection from public sources about the operation of open source projects using tools such as Prospector.

In addition to the formal metrics that the organization is developing, CHAOSS has also published a list of broad categories of indicators that they've heard repeatedly over the course of their research.

Unsurprisingly, these include a variety of measures related to community growth, momentum, and breadth of code contributions beyond the project's creator (whether an individual or an organization). However, there's also a theme around the role of maintainers. How timely are they? How attentive to users? This comes back to making contributions easier.

A Broader View of Health

Projects also don't stand alone. The health of their ecosystem—upstream, downstream, and related projects—matters too. What's the *aggregate* project-tree health, which is to say the combined health metrics of all linked dependencies? An individual project may be healthy but if it's heavily dependent on other projects that aren't, it should correct the issues in the dependencies or switch away from them. Doing so may be fairly straightforward in the case of a library for which there are a variety of reasonable substitutes. It will be much more difficult for a project that's effectively a satellite orbiting a much larger project that's failing. (Conversely, a large successful project like Kubernetes can energize a wide range of smaller efforts that complement or expand the functionality of the mothership in various ways.)

Finally, any approach to measurement and metrics ultimately depends on the context of a given project. For example, mature projects might appear inactive if no new features are being developed even though the community remains responsive to bug reports and security issues. But that may just mean the project has attained its goal and the community sees no need to add features for the sake of adding features.

Important metrics for community and project health also tend to shift over time. A given project may go through a growth spurt where a lot of attention should be devoted to how effectively new contributors are being onboarded and mentored. Later in life, the

same project may shift more focus to how well it's doing moving existing contributors into leadership roles as current maintainers inevitably move on.

The nature of different projects also tend to make certain tasks easy and some hard. A project started within one company and then released as open source will often have to work hard to attract external contributions at all. This makes tracking the diversity of contributions—and putting the processes and code in place to simplify those contributions—paramount. In other cases, diversity comes naturally because of the way the project was formed and collaboration among parties with sometimes divergent interests becomes a more interesting metric.

If all this sounds complicated, it is. And neither researchers nor expert practitioners have been able to reduce measurement of software development practices and community health to a template or a science. However, they have identified at least some approaches and ideas to increment on and some bad practices to avoid. As Kaner and Bond write in the conclusion of their 2004 paper: “There are too many simplistic metrics that don’t capture the essence of whatever it is that they are supposed to measure. There are too many uses of simplistic measures that don’t even recognize what attributes are supposedly being measured. Starting from a detailed analysis of the task or attribute under study might lead to more complex, and more qualitative, metrics, but we believe that it will also leads to more meaningful and therefore more useful data.”

Reflecting and Informing

On the one hand, much of what we know about open source software development is relatively new. Most research specifically into open source development and community practices is only about a decade or so old. Furthermore, what research exists from the early days of open source is arguably of limited relevance to today when open source is so pervasive and often commercialized.

However, past research into fields as diverse as invention and innovation, collaboration, motivation, and measurement have produced learnings which are often relevant to open source projects and communities. At the same time, precisely because open source software development happens in the open, it provides a rich data trove for new research into these topics as well.

Open source software is a test bed for how people (and organizations) work together, why people work together, and the types of measurements that we should be thinking about. Much of this is an art. But there's also science. Which is why formal research is at least worth a look.

CHAPTER 5

Business Models

When many people first hear about businesses that make their living selling open source software, their first question is usually something like “How can you sell something that’s available for free?” That complicated answer is addressed in this chapter.

Even restricting the discussion to “free as in beer” doesn’t simplify it much. The business world is full of examples of free and paid services or products supporting and subsidizing each other in various ways. It’s a hard formula to get right, and business models based on such approaches can rapidly shift with technology and consumer habits. Ask any newspaper publisher.

Indeed, it’s misleading, or at least simplistic, to refer to an “open source business model” as if open source software existed outside of normal commercial relationships in which money is traded for a product or service of value.

Which brings us to the other overarching theme that I’ll cover. Open source software and the forces that have helped to shape it don’t exist in a bubble. They’re part of broader trends in software development, in business interactions, and in organizational culture generally.

How Can You Sell Something That You Give Away?

Free is a multifaceted word. Its meaning is less muddled in some languages. Latin and some languages deriving from it use different concepts for the concept of freedom (*liber* in Latin) and zero price (*gratis* in Latin). “Free” in modern English appears to have come from the Old English/Anglo-Saxon side of its language tree where *freagon* came to embody both meanings. (English also has language roots in Latin by way of Norman French.)

As we've seen, "free software" in the sense that we've been discussing here has always been about free as in freedom rather than free as in beer. As Stallman writes in the first footnote to the GNU Manifesto:

The wording here was careless. The intention was that nobody would have to pay for permission to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between "free" in the sense of freedom and "free" in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies—and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.

Freedom doesn't pay the bills though.

The GNU Manifesto offers up some possibilities that apply mostly to individuals. For example, "People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services." Stallman also makes some more questionable suggestions such as "Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF [National Science Foundation] to spend on software development." None of these really look like scalable business models.

Is There an "Open Source Business Model"?

In *Free: The Future of a Radical Price* (Hyperion, 2009), former *Wired* Magazine editor-in-chief Chris Anderson refers to "free" as the "most misunderstood word" and describes many of the ways in which giving things away gratis can be profitable. In all, he describes 50 business models that fall into three broad categories.

Categories of Business Models

There are direct subsidies. For example, Apple gives away many types of Apple Store Genius Bar tech support as part of the package you get when you buy an Apple phone or computer.

There are three-party or two-sided markets in which one customer class subsidizes another. Ad-supported media, including companies like Facebook, fall broadly into this category in that they're giving away a service to consumers while charging businesses for access to that audience.

Finally, there's freemium. A certain class of service or product is free but you need to pay to upgrade. Freemium is a common approach for selling many types of software. In-app purchases on iPhone and Android smartphones are classic examples. You can download the basic app for free, but you need to pay money to remove ads or get more features.

Getting the Balance Right

A general challenge with freemium is getting the balance of free and paid right. Make free too good and your conversion rate to paid might not be high enough to be profitable—even among people who would have been willing to pay if they had to in order to use the product at all. I use a variety of programs and software services that are useful enough to me that I'd probably be willing to pay for them if necessary. However, the free tier meets my needs well enough; I may not even value the incremental features of the paid version at all.

On the other hand, cripple the free version too much and it becomes uninteresting in its own right. If this happens, you don't get many people to try your software.

Take the example of a not-so-hypothetical online storage provider trying to grow its customer base. They could offer a free trial. That has pros and cons but probably isn't a good fit here. (Users will upload things and some of them will lose their only copies of those things when the trial ends. They'll be mad. Probably not a great plan.) Instead, you decide that users will be able to sign up to get some storage that they can keep using forever at no charge. But if they want more storage, they'll have to pay some tiered monthly fee depending upon how much they want. But how much should you give them?

You could be a cheapskate and give them 10 megabytes. Seriously? You can store one MP3 song with that amount. No one's going to bother to use that. OK. How about 10 terabytes—a million times as much? That's more storage than all but a tiny sliver of what individuals need. You'll get lots of sign-ups (assuming the service is otherwise useful) but few paying customers who require more space.

Building the Funnel with Free

The nice thing about a freemium model is that it's a good way to acquire users with the product itself. They're not *paying* customers yet, but they're well into your sales funnel in marketing-speak. (To be specific, they're potentially in an evaluation phase, which is often listed as the third phase of the funnel after awareness and interest.) They're using your product. They still have to like it. And they still have to decide to buy it. But getting a potential customer to evaluate your product is a big step. Freemium models for software, especially relatively simple-to-use software, make getting from initial awareness to evaluation a relatively quick and low friction process if the experience for new users is otherwise solid.

At one level, business models that include open source software can be thought of as variants on freemium. However, be careful with this framing. It can encourage simplistic thinking. Successful business models usually involve more than just charging for support or offering consulting as an option. Furthermore, approaches such as open core may look like they're open source on the surface without benefiting much from the open source development model.

Open Core versus Open Source

With open core, a company gives away a free product that is open source but then sells additional proprietary software that complements it in some way. This often takes the form of something like a “community” edition that's free and an “enterprise” edition that requires either a license or a subscription fee. A typical distinction is that the enterprise edition will include features that tend to be important for large organizations running software in production but aren't as big a deal for individuals or casual use. Andrew Lampitt is credited with coining the open core term.

The MySQL database—acquired by Oracle when it bought Sun—is a typical case. MySQL Enterprise Edition “includes the most comprehensive set of advanced features, management tools and technical support to achieve the highest levels of MySQL

scalability, security, reliability, and uptime. It reduces the risk, cost, and complexity in developing, deploying, and managing business-critical MySQL applications.” Thus, even though you can use the base MySQL project for free, many of the features that you probably want as an enterprise user are behind the paywall.

In part because the upsell features are often not clearly partitioned off from the core project, many open core products require their contributors to sign a contributor license agreement (CLA), which assigns rights to the commercial owner of the product. (It may or may not assign copyright but, in any case, it gives the owner the right to use the contributions under a proprietary license if they want to.) Pure open source projects may use CLAs as well, but in that case, they serve a somewhat different purpose. For example, the Eclipse Contributor Agreement gives as its rationale: “It’s basically about documenting the provenance of all of the intellectual property coming into Eclipse. We want to have a clear record that you have agreed to the terms under which the Eclipse community has agreed to accept contributions.”

Many vendors are attracted to open core because it’s effectively a proprietary business model that uses open source but isn’t itself really a business model directly based on open source. What’s being sold is the proprietary add-ons. The vendor’s hope is that they’ve gotten the free and paid balance right. That the free is good enough to attract users and even outside contributors. But that most customers who would have been willing and able to pay anyway will pony up for the premium version.

As the president of the OSI, Simon Phipps, writes: “Open core is a game on rather than a valid expression of software freedom, because it does not [provide] software freedom for the software user . . . to use the package effectively in production, a business probably won’t find the functions of the core package sufficient, even in the (usual) case of the core package being highly capable. They will find the core package largely ineffective without certain ‘extras,’ and these are only available in the ‘enterprise version’ of the package, which is not open source. To use these features, you are forced to be a customer only of the sponsoring company. There’s no alternative, no way to do it yourself if the value delivered doesn’t justify the expense involved, or if you are time-rich and cash-poor. Worse, using the package locks you in to the supplier.”

Are You Taking Advantage of Open Source Development?

That’s the perspective from the users of the software. But what about from the vendor’s perspective? Is this just an argument that open core is not a sufficiently ideologically pure approach to building a business based on open source?

The issue isn't ideological purity. It's that a business model that's not fully based on open source doesn't accrue the full benefits of being based on open source either.

From a customer's perspective, if you need the enterprise features, you need the proprietary product. The fact that there's an open source version lacking features you need isn't all that relevant. It's no different from a freemium approach to traditional proprietary software or software-as-a-service. You can't try out what you don't have access to. The power of freemium to get potential users in the door can be significant. It's just that open core isn't uniquely different from proprietary approaches to selling software just because open source is part of the mix.

Companies also find that an open core model often doesn't bring the full benefits of the open source development model. The power of open source as a development approach isn't that anyone can see your code. It's that individuals and companies can collaborate and work together cooperatively to make better software. However, open core almost can't help sending off a vibe that the vendor owns the open source project and its community given that proprietary extensions depend on the open source core. It can be hard to attract outside contributors in this situation—which can be a community management challenge under the best of circumstances. The result is often an open source project that is open source in name only.

Subscriptions and Support

There's another freemium model that's common in open source. In fact, it's often called out as *the* open source business model although that isn't really correct or is, at least, an oversimplification. With this model, you can obtain and use fully functional, nothing-held-back software under an open source license. You can use it without paying for as long as you want, no strings attached. But, if you want support you're going to need to pay for it in some form. This differs from the typical subscription arrangement for proprietary software—Adobe Creative Cloud, for example—with which you lose access to the software if your subscription lapses.

It's also different from the historical approach to proprietary software, which combined an up-front software license with some sort of maintenance fee for minor updates and support.

The Rise of the Independent Software Vendor

It's difficult to identify the first company to sell software that wasn't also hawking hardware (which is to say, the first Independent Software Vendor (ISV)). However, Cincom Systems—founded in 1968—is a good candidate. It sold what appears to be the first commercial database management system not to be developed by a system maker like IBM. Fun fact: not only is Cincom still extant as a private company in 2018 but one of its founders, Thomas Nies, is the CEO.

Over time, pure-play or mostly pure-play software companies packaging up bits and selling them became the dominant way in which customers acquired most of their software. As we've seen, ISVs like Microsoft selling closed-source proprietary software even became major suppliers of the operating systems and other “platform” software that historically were supplied by vendors as part of a bundle with their hardware.

When open source software came onto the scene, it didn't bring with it the same requirement to purchase an up-front license. However, many users still wanted the other benefits associated with having a support relationship with a commercial entity.

Open Source Support Arrives

Probably the first company to systematically provide support for open source software in a formal way was Cygnus Solutions. It was founded by John Gilmore, Michael Tiemann, and David Henkel-Wallace in 1989 under the name Cygnus Support. Its tagline was: Making free software affordable.¹

Cygnus Solutions maintained a number of the key GNU software products, including the GNU Debugger. It was also a major contributor to the GCC project, the GNU C compiler.

As Tiemann described the company in 1999's *Open Sources: Voices from the Open Source Revolution* (O'Reilly Media): “We wrote our first contract in February of 1990, and by the end of April, we had already written over \$150,000 worth of contracts. In May, we sent letters to 50 prospects we had identified as possibly interested in our support, and in June, to another 100. Suddenly, the business was real. By the end of the first year, we had written \$725,000 worth of support and development contracts, and everywhere we looked, there was more opportunity.”

¹<http://www.oreilly.com/openbook/opensources/book/tiemans.html>

In the same book, Tiemann also touches on something else that would come to be important to successful open source businesses when he wrote: “Unless and until a competitor can match the 100+ engineers we have on staff today, most of whom are primary authors or maintainers of the software we support, they cannot displace us from our position as the ‘true GNU’ source (we supply over 80% of all changes made to GCC, GDB, and related utilities).”

Shortly after Tiemann wrote those words, Red Hat—which had just gone public in August 1999—acquired Cygnus. Red Hat dates back to 1993 when Bob Young, incorporated as the ACC Corporation, started a mail-order catalog business that sold Linux and Unix software accessories out of his home. The following year, Marc Ewing started distributing his own curated version of Linux and he chose Red Hat as the name. (He picked that unusual name because he was known for wearing his grandfather’s red Cornell lacrosse hat when he worked at his job helping fellow students in the computer lab at Carnegie Mellon.)

Young found himself selling a lot of copies of Ewing’s product. In 1995, they joined together to become Red Hat Software, which sold boxed copies of Red Hat Linux, an early Linux distribution.

Linux Distributions Appear

Distributions, or “distros” as they’re often called, first appeared in 1992 but more active projects arrived the next year. That’s when Patrick Volkerding released Slackware based on the earlier but not well-maintained SLS. The year 1993 also saw Ian Murdoch’s founding of Debian and its release near the end of the year.

Distributions brought together the core operating system components, including the kernel, and combined them with the other pieces, such as the utilities, programming tools, and web servers needed to create a working environment suitable for running applications. Distributions were a recognition that an operating system kernel and even the kernel plus a core set of utilities (such as those that are part of GNU in the case of Linux) aren’t that useful by themselves.

Over the next decade, the number of distributions exploded although only a handful were ever sold commercially.

Support was one of the first things to get added to commercial Linux distributions. Initially, this meant pretty much what it did with traditional retail boxed software. You called a helpdesk if you were having trouble installing something, the software didn’t work as promised, or you wanted to report a bug. However, thinking about what a

commercial software vendor like Red Hat does as support for open source software is not only too narrow a view. It's not the right lens.

Subscriptions: Beyond Support

Rather, as Steven Weber writes, you should be thinking about “building profitable economic models around the open source process.”

In Red Hat's case, it's an enterprise subscription software business that is based on an open source development model. What does this subscription look like in the context of open source software? It developed over time. It came about through experimenting, innovating, and perfecting a community-based model. It came through experiencing how to best participate in communities; adding features and functionality desired by customers; and then testing, hardening, compiling, and distributing stable, workable versions to customers.

One of the things that Michael Tiemann wrote back in 1999 that's still very relevant today is that part of a business model for open source software is establishing in-house expertise in the design, optimization, and maintenance of the products being sold. This may be an obvious point in the case of proprietary software that is written by a single company. However, with open source software, it's also difficult to provide effective support in the absence of active participation in the communities developing the software. That participation is what leads to having the expertise to solve difficult support problems.

And it goes beyond support. Users of software often want to influence product direction or the development of new features. With open source software, users can do so directly. However, working with the communities in which the software is developed isn't necessarily easy or obvious to a company that isn't familiar with doing so. As we've seen, there's not really a template. Communities have different governance models, customs, and processes. Or just quirks. Even organizations that do want to participate directly in setting the direction of the software they use can benefit from a guide.

Focusing on Core Competencies

Furthermore, many organizations don't want to (or shouldn't) spend money and attention on developing all the software that they use. When I was an industry analyst in the early 2000s, I would talk with the banks and other financial institutions who were among the earliest adopters of Linux after the Internet infrastructure providers.

Large banks had technologists with titles like “director of kernel engineering.” But here’s the thing. Banks are not actually in the business of writing and supporting operating systems. They need operating systems. But they also need bank branches. Yet they’re not in the construction business.

Over time, banks and other end users did increasingly participate in community-based open source development. We saw the example of AMQP development, for example. However, especially for platforms that make up their infrastructure, most enterprises prefer to let companies that specialize in the software do much of the heavy lifting.

Ultimately, subscribers can choose the degree to which they participate in and influence technology and industry innovation. They can either use the open source product as they would any other product. Or they can actively participate in setting the development direction to a degree that is rare with proprietary products.

Open source software subscriptions do indeed provide fixes, updates, assistance, and certifications in a way that doesn’t look that different from other commercial software products. And that may be enough for many customers. However, the ability to participate in the open source development model creates opportunities that don’t exist with proprietary software.

Aligning Incentives with Subscriptions

Furthermore, subscriptions create different incentives for vendors than up-front licenses do. The usual way that licenses work is that there’s an up-front license fee and upgrade fees for major new versions, as well as ongoing maintenance charges. As a result, there’s a strong incentive for vendors to encourage upgrades. In practice, this means that—while the company selling the software has contractual obligations to fix bugs and patch security holes—they would actually prefer that customers upgrade to new versions when they become available. There’s thus an active disincentive to add new features to existing software.

With a subscription model, on the other hand, so long as a customer continues to subscribe, it doesn’t matter so much to the vendor whether a customer upgrades to a new version or not. There’s still some incentive to get customers to upgrade. It takes effort to add new features to older versions and, at some point, it can just become too hard to continue providing support. But there’s no financial impetus creating an artificial urgency to force upgrades.

Subscriptions do sometimes get a bad rap, especially in proprietary consumer software and services. But that's mostly because, with most subscriptions of this type, you only retain access to the software so long as you pay the subscription fee. For someone only running some piece of software occasionally and casually, that can be a bad deal compared to just using a five-year-old software package that's no longer supported but works fine for the task at hand.

Open source subscriptions are different though because you retain full control over and access to software even if you let a subscription lapse. That's fundamental to free and open source software. It's yours to do with as you please. That's at the heart of a business model that makes profitable companies built around open source possible.

From Competition to Coopetition

The rise of open source software has paralleled other changes taking place in the software industry and beyond. Some of these changes arguably take their cues from the open source development model. Others are more likely the result of some of the same influences that helped make the widespread adoption of open source software possible such as the Internet and inexpensive computers.

One of the broad changes that has paralleled the rise of open source is an increasing trend toward greater coopetition—cooperative competition.

Coopetition Gets Coined

The term dates back to the early 20th century, but it started to see widespread use when Novell's Ray Noorda began using the term to describe the company's business strategy in the 1990s. For example, Novell was at the time planning to get into the Internet portal business, which required it to seek partnerships with some of the same search engine providers and other companies that it would also be competing with.

In 1996, Harvard Business School's Adam Bradenburger and Yale's Barry Nalebuff wrote a *New York Times* best-selling book on the subject, adopting Noorda's term and examining the concept through the lens of game theory. They described it as follows. "Some people see business entirely as competition. They think doing business is waging war and assume they can't win unless somebody else loses. Other people see business entirely as cooperative teams and partnerships. But business is both cooperation and competition. It's coopetition."

The basic principles have been around forever. Marshall University's Robert Deal describes in *The Law of the Whale Hunt: Dispute Resolution, Property Law, and American Whalers, 1780–1880* (Cambridge University Press, 2016) how “Far from courts and law enforcement, competing crews of American whalers not known for their gentility and armed with harpoons tended to resolve disputes at sea over ownership of whales. Left to settle arguments on their own, whalemens created norms and customs to decide ownership of whales pursued by multiple crews.”² Many situations aren't ruled solely by either ruthless competition or wholly altruistic cooperation.

Why Coopetition Has Grown

The theory behind coopetition isn't that well established with the result that there's debate over where coopetition is most effective and what the most effective strategies are. However, a 2012 paper by Paavo Ritala notes that “it has been suggested that it occurs in knowledge-intensive sectors in which rival firms collaborate in creating interoperable solutions and standards, in R&D, and in sharing risks.”³

That's a good description of the IT industry, but it's an increasingly good description of the *many* industries that are increasingly selling products and services that are enabled by software or simply are software. “Software is eating the world” as venture capitalist (and co-author of the first widely used web browser) Marc Andreessen famously put it in a 2011 *Wall Street Journal* piece. It seems at least plausible that coopetition's high profile of late is the result of complexity levels and customer demands that make it increasingly difficult to successfully avoid cooperation.

By way of contrast, I still remember one day in the early 1990s. I got an email from a sales rep, livid because he had learned that a networking card in a new computer system we had announced was made by Digital Equipment, a major competitor. Among the rep's choice words were something along the lines of “I'm fighting with these guys every day and you go and stab me in the back.”

I tell this story because it nicely illustrates the degree to which the computer systems market has changed. Today, the idea that having commercial relationships with a competitor to supply some part or service would be scandalous or even especially notable would seem odd under most circumstances. There are echoes of this sort of

²<http://legalhistoryblog.blogspot.com/2016/04/deals-law-of-whale-hunt.html>

³<https://onlinelibrary.wiley.com/doi/epdf/10.1111/j.1467-8551.2011.00741.x>

behavior in the scuffles between Apple, Google, and Amazon in smartphones and voice assistants. But those are notable mostly because they're not really the norm.

Coopetition is at the heart of most larger open source projects in which the participants are mostly developers working on the software as part of their day jobs. Look through the top contributors to the Linux kernel and you'll see multiple semiconductor companies, software companies with Linux distributions, computer system vendors, and cloud service providers.⁴ Companies within each of these groups are often direct competitors and, indeed, may compete with others as well in certain aspects of their business. When the OpenStack Foundation was created, it was in large part to explicitly create a structure that could accommodate participation by competing corporate interests.

Open Source: Beneficiary and Catalyst

Open source software development has both benefited from and been a catalyst for coopetition. Seeing companies working cooperatively on an open source project, it's easy to dismiss the novelty of working together in this way. After all, companies have cooperated in joint ventures and other types of partnerships forever.

What we observe with open source projects, however, is a sharply reduced level of overhead associated with cooperation. Companies work together in a variety of ways. But many of those ways involve contracts, non-disclosure agreements, and other legal niceties. While open source projects may have some of that—contributor license agreements for example—for the most part, starting to work on a project is as simple as submitting a pull request to let others know that you have pushed code to a repository.

Extensive involvement in a major project tends to be more formal and more structured of course. The details will depend on the project's governance model, but major project contributors should be on the same page as to the project's direction. Nonetheless, the overall process for working together in an open source project tends to be lighter weight, lower overhead, and faster than was historically the case for companies working together.

One specific change in this vein that we've seen is the way that software standards are now often developed.

⁴<https://www.linuxfoundation.org/2017-linux-kernel-report-landing-page/>

Coopetition and Standards

Typically, we talk about two types of standards. One type is *de jure* standards, or standards according to the law. These are what you get when industry representatives, usually including competitors, sit down as part of a standards organization or other trade organization to create a standard for something. The process can be very long and arduous. And also infamous for often producing long and technically rigorous, in an academic way, specifications that don't actually get used much in the real world.

The Open Systems Interconnection model (OSI model) is one example. While OSI's conceptual seven-layer model has been widely adopted as a way to talk about layers of the networking software stack, software that directly implemented OSI was never much used.

By contrast, TCP/IP, Transmission Control Protocol (TCP) and the Internet Protocol (IP), came out of research and development conducted by the Defense Advanced Research Projects Agency (DARPA) in the late 1960s. Today, they're the core communication protocols used by the Internet. Although TCP/IP was subsequently ratified as a formal standard, it started out as a *de facto* standard by virtue of widespread use. Widely used proprietary products such as Microsoft Windows or x86 processors can also be viewed as *de facto* standards.

Open source has developed something of a bias toward a *de facto* standardization process that is effectively coopetition and standardization through code. Or, if you prefer, "code first."

We've seen this play out repeatedly in the software containers space. While they were subsequently standardized under the Open Container Initiative, image runtime and image formats existed as implementations before they became a standard. The same is true of container orchestration with Kubernetes evolving to be the most common way of orchestrating and managing a cluster of containers. It's a standard based on the size of its community and its adoption rather than the action of a standards body.

This approach is very flexible because it allows companies to work together developing software and then iterate as needed. There's another advantage as well. One of the problems with specifications is that they rarely fully specify everything. As a result, software that's based on standards often has to make assumptions about unspecified details and behaviors. By contrast, a standard achieved through a code first approach is its own reference implementation. It's therefore a more effective approach to coopetition than developing a specification in committee only to have parties go off and develop their individual implementations—which can be subtly incompatible as was the case with the Fibre Channel storage interconnect early on, to give one example.

The Need for Speed

The ascendancy of the open source development model as an approach for collaboration and innovation wasn't the only interesting IT trend taking place in the mid- to late 2000s. A number of things were coming together in a way that would lead to both new platforms and new development practices.

From Physical to Virtual

Server virtualization, used to split physical servers into multiple virtual ones, was maturing and IT shops were becoming more comfortable with it. Virtualization was initially intended to reduce the number of boxes needed and hence to cut costs. However, it came to have other uses as well. Ubiquitous virtualization meant that IT organizations were becoming more accepting of not knowing exactly where their applications are physically running. In other words, another level of abstraction was becoming the norm as has happened many times in many places over the history of computing.

A vendor and software ecosystem was growing out alongside and on top of virtualization. One specific pain point this ecosystem addressed was in the area of “virtualization sprawl,” a problem brought about by the fact that virtualization made it so easy to spin up new systems that the management burden could get out of hand. Concepts like automation, policy-based administration, standard operating environments, and self-service management were starting to replace system admin processes that had historically been handled by one-off scripts—assuming they weren't simply handled manually.

The Consumerization of IT

IT was also consumerizing and getting more mobile. By 2007, many professionals no longer used PCs tethered to a local area network. They used laptops running on Wi-Fi. Then Apple introduced the iPhone. Soon smartphones were everywhere, usually purchased by employees even though they were often used for both personal and business purposes. Meanwhile, on the software side, users were getting accustomed to responsive, slick consumer web properties like Amazon and Netflix during this post-dot-com Phase 2 of the web. Stodgy and hard-to-use enterprise software looked less attractive than ever.

Line of business users in companies also started noticing how slow their IT departments were to respond to requests. Enterprise IT departments rightly retorted that they operate under a lot of constraints—whether data security, detailed business requirements, or uptime—that a free social-media site does not. Nonetheless, the consumer web increasingly set an expectation and, if IT couldn't or wouldn't meet it, users would go to online services—whether to quickly put computing resources on a credit card or to purchase access to a complete online application.

Even those enterprise IT shops with tightly run software practices could see that the speed at which big Internet businesses such as Amazon and Netflix could enhance, update, and tune their customer-facing services was at a different level from what they could do. Yet a miniscule number of these deployments caused any kind of outage. These companies were different from more traditional businesses in many ways. Nonetheless they set benchmarks for what is possible.

Which brings us to DevOps.

The Rise of DevOps

DevOps touches many different aspects of the software development, delivery, and operations process. But, at a high level, it can be thought of as applying open source principles and practices to automation, platform design, and culture. The goal is to make the overall process associated with software faster, more flexible, and incremental. Ideas like the continuous improvement based on metrics and data that have transformed manufacturing in many industries are at the heart of the DevOps concept. Amazon and Netflix got to where they are in part by using DevOps.

The DevOps Origin Story

DevOps grew out of Agile software development methodologies, which were formally laid out in a 2001 manifesto⁵ although they had roots going back much further. For example, there are antecedents to Agile and DevOps in the lean manufacturing and continuous improvement methods widely adopted by the automobile industry and elsewhere. The correspondence isn't perfect; lean approaches focus to a significant degree on reducing inventory, which doesn't cleanly map to software development.

⁵<http://agilemanifesto.org/>

Nonetheless, it's not hard to find echoes of principles found in the Toyota Way (which underlies the Toyota Production System) like “respect for people,” the right process will produce the right results,” and “continuously solving root problems” in DevOps. Appreciating this lineage also helps to understand that, while appropriate tools and platforms are important, DevOps is at least equally about culture and process.

The DevOps term was coined by Belgian consultant Patrick Debois who had been frustrated about the walls of separation and lack of cohesion between application methods and infrastructure methods while on an assignment for the Belgian government. A presentation by John Allspaw and Paul Hammond at the O'Reilly Velocity 09 conference entitled “10 Deploys a Day: Dev and Ops Cooperation at Flickr” provided the spark for Debois to form his own conference called Devopsdays in Ghent, Belgium in 2009 to discuss these types of issues. DevOpsDays have since expanded as a sort of grassroots community effort to the point where, in 2018, there are dozens held every year around the world.

According to Frederic Paul in an InfoQ video interview from April 2012, Debois admitted that naming the movement was not as intentional as it might seem: “I picked ‘DevOpsDays’ as Dev and Ops working together because ‘Agile System Administration’ was too long,” he said. “There never was a grand plan for DevOps as a word.”⁶

Another noteworthy DevOps moment was the publication of *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, written by Gene Kim, Kevin Behr and George Spafford in 2013 (IT Revolution Press). This book is a sort of fable about an IT manager who has to salvage a critical project that has bogged down and gotten his predecessor fired. A board member mentor guides him through new ways of thinking about IT, application development, and security—introducing DevOps in the process. Although DevOps has both evolved and been written about more systematically since then (including *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* by Gene Kim, Patrick Debois, Jez Humble, and John Willis (IT Revolution Press, 2016)), *The Phoenix Project* remains an influential text for the movement.

⁶<https://blog.newrelic.com/2014/05/16/devops-name/> Debois actually advocates for capitalizing the term as Devops rather than DevOps.

DevOps: Extending Beyond Agile

DevOps widened Agile principles to encompass the entire application life cycle including production operations. Thus, operations and security skills needed to be added to the cross-functional teams that included designers, testers, and developers. Improving collaboration, communication, and the level of cross-functional skills is an important DevOps tenet.

Taken to an extreme, there might even no longer be devs and ops people, but DevOps skill sets. But, more commonly, this view of DevOps focuses on “two pizza” cross-functional teams—small, multidisciplinary groups that own a service from its inception through its entire life cycle. This works in part because such services are autonomous, have bounded context, and can be developed independent of other services and groups, so long as they honor their API contract. It also assumes that these “generalist” teams have the necessary skills to operate the underlying platform.

For example, when thinking about security as part of DevOps—or even using the DevSecOps term to remind us of its importance—developers (and operations) don’t suddenly need to become security specialists in addition to the other hats they already wear. But they can often benefit from a greater awareness of security best practices (which may be different from what they’ve become accustomed to) and shifting away from a mindset that views security as some unfortunate obstacle.

Abstractions to Separate Concerns

However, especially in larger organizations, DevOps has evolved to mean something a bit different than closely communicating cross-functional teams, developers on pager duty, or sysadmins writing code. Those patterns may still be followed to greater or lesser degrees, but there’s a greater focus on clean separation of concerns. It’s about enabling ops to provide an environment for developers, then get out of the way as much as possible.

This is what Adrian Cockcroft, Netflix’s former cloud and DevOps guru—he’s now at Amazon Web Services (AWS)—was getting at with the “No Ops” term when he wrote about it.⁷ While Netflix was and is a special case, Cockcroft hinted at something that’s broadly applicable: In evolved DevOps, a lot of what ops does is put core services in place and get out of the way. There’s value in creating infrastructure, processes, and tools

⁷<http://perfcap.blogspot.com/2012/03/ops-devops-and-noops-at-netflix.html>

in a way that devs doesn't need to interact with ops as much—while being even more effective. (Netflix largely operated using Amazon cloud services, so they had very little infrastructure they operated themselves, in spite of their vast scale.)

Reducing the friction of interactions between devs and ops doesn't always mean making communication easier. It can also involve making communication unnecessary. Think about it this way: You do not, in fact, want to communicate with a bank teller more efficiently. You want to use an ATM. You want self-service.

With this model of DevOps, key aspects of operations happen outside of and independent of the application development process.

Of course, communication between dev and ops (as well as other disciplines) still matters. The most effective processes have continuous communication. This enables better collaboration, so that teams can identify failures before they happen; feedback, to continuously improve and cultivate growth; and transparency.

Site Reliability Engineers

At this point, it's worth mentioning Site Reliability Engineering. The term came out of Google in about 2003 when a team led by Ben Treynor was tasked to make Google's sites run smoothly, efficiently, and more reliably. Like other companies with large-scale infrastructures, Google was finding that existing system management paradigms didn't provide either the reliability or the ability to deploy new features quickly that they needed.

The idea is that a site reliability engineer (SRE) will spend about half their time on ops-related tasks like manual interventions and clearing issues. However, because the goal is to make the underlying system as automated and self-healing as possible, an SRE also spends significant time writing software that reduces the need for manual interventions or adds new features. Conceptually, this is somewhat like how a traditional system admin would write a script after they had to do the same task a few times. But the SRE concept puts that practice on steroids and expands the ops role into one with a much larger software development component.

Google's Seth Vargo and Liz Fong-Jones argue that SRE is a variant of DevOps or "DevOps is like an abstract class in programming, and SRE is one possible implementation of that class" as they put it. I think of it more as an evolved form of ops for a separation-of-concerns DevOps model given that SRE teams support the groups actually developing software services. An SRE approach may indeed shift the location

of the boundary between ops-centric roles and dev-centric roles. A concrete example might be one which embeds an application's operational domain knowledge for a cluster of containers.⁸ But I'd argue that it's still effectively a specialized ops function.

Manufacturing Analogs

Like DevOps more broadly, separation of functions also hearkens back to earlier examples from manufacturing and industrial organization. Red Hat's Matt Micene writes that "The 'Dev' and 'Ops' split is not the result of personality, diverging skills, or a magic hat placed on the heads of new employees; it's a by-product of Taylorism and Sloanianism. Clear and impermeable boundaries between responsibilities and personnel is a management function coupled with a focus on worker efficiency. The management split could have easily landed on product or project boundaries instead of skills, but the history of business management theory through today tells us that skills-based grouping is the 'best' way to be efficient."⁹

In any case, DevOps should be viewed as a set of principles rather than a prescriptive set of rules.

Open Source and DevOps

Open source relates to DevOps across aspects that include platforms and tooling, process and automation, and culture.

Platforms and Tooling

A DevOps approach can be applied on just about any platform using any sort of tooling. DevOps can even be a good bridge between existing systems, existing applications, and existing development processes and new ones. The best tools in the world also won't compensate for broken processes or toxic culture. Nonetheless, it's far easier to streamline DevOps workflows with the right platform and tools.

Open source tooling is the default in DevOps. A 2015 DevOps Thought Leadership Survey by market researcher IDC found that a whopping 82 percent of early DevOps

⁸<https://coreos.com/blog/introducing-operators.html>

⁹<https://opensource.com/open-organization/17/5/what-is-the-point-of-DevOps>

adopters said open source was “a critical or significant enabler of their DevOps strategy.” What's more, the further along the survey respondents were in implementing DevOps initiatives, the more important they thought open source and DevOps open source tools were.

At the platform level, a key trend pushing the use of new technologies is a shift from static platforms to dynamic, software-defined platforms that are programmable, which is to say controllable through APIs. The OpenStack project is a good example of how software-defined storage, software-defined networking, identity management, and other technologies can come together as a complete programmable infrastructure.

Containers are another important element of modern distributed application platforms. Containers modernize IT environments and processes, and provide a flexible foundation for implementing DevOps. At the organizational level, containers allow for appropriate ownership of the technology stack and processes, reducing hand-offs and the costly change coordination that comes with them. This lets application teams own container images, including all dependencies, while allowing operations teams to retain full ownership of the production platform.

With a standardized container infrastructure in place, IT operations teams can focus on building out and managing clusters of containers, meeting their security standards, automation needs, high availability requirements, and ultimately their cost profiles.

When thinking about the tool chain associated with DevOps, a good place to start is the automation of the continuous integration/continuous deployment (CI/CD) pipeline. The end goal is to make automation pervasive and consistent using a common language across both classic and cloud-native IT. For example, Ansible allows configurations to be expressed as “playbooks” in a data format that can be read by both humans and machines. This makes them easy to audit with other programs, and easy for non-developers to read and understand.

A wide range of other open source tools are common in DevOps environments including code repositories like Git, monitoring software like Prometheus and Hawkular, logging tools like Fluentd, and container content tools like Buildah.

Process

We also see congruence of open source development processes and those of DevOps. While not every open source project puts in the up-front work to fully implement DevOps workflows, many do.

For example, Edward Fry relates the story of one community that found “there are some huge benefits for part-time community teams. Planning goes from long, arduous design sessions to a quick prototyping and storyboarding process. Builds become automated, reliable, and resilient. Testing and bug detection are proactive instead of reactive, which turns into a happier clientele. Multiple full-time program managers are replaced with self-managing teams with a single part-time manager to oversee projects. Teams become smaller and more efficient, which equates to higher production rates and higher-quality project delivery. With results like these, it’s hard to argue against DevOps.”¹⁰

Whether or not they check all the DevOps boxes, significant open source projects almost can’t help but to have at least some characteristics of a DevOps process.

For example, there needs to be a common and consistent view into code. DevOps and open source projects are both well-adapted to using a distributed approach whereby each developer works directly with his or her own local repository with changes shared between repositories as a separate step. In fact, Git, which is widely used in platforms for DevOps, was designed by Linus Torvalds based on the needs of the Linux kernel project. It’s decentralized and aims to be fast, flexible, and robust.

And remember the earlier discussion about creating a good experience for new contributors by providing them with rapid feedback and incorporating their code when it’s ready? Automation and CI/CD systems are a great way to automate testing, build software more quickly, and push out more frequent releases.

Iteration, Experimentation, and Failure

At a higher level, DevOps embraces fast iteration, which sounds a lot like the bazaar approach to software development. They don’t align perfectly; software developed using a DevOps approach can still be carefully architected. However, DevOps has a general ethos that encompasses attributes like incremental changes, modularity, and experimentation.

Let’s talk about experimentation a bit more. Because it has a flip side. Failure.

Now that’s a word with a negative vibe. Among engineering and construction projects, it conjures up the *Titanic* sinking, the Tacoma Narrows bridge twisting in the wind, or the space shuttle *Challenger* exploding. These were all failures of engineering design or management.

¹⁰<https://opensource.com/article/18/4/devops-compatible-part-time-community-teams>

Most failures in the pure software realm don't lead to the same visceral imagery as the above, but they can have widespread financial and human costs all the same. Think of the failed [Healthcare.gov](http://www.healthcare.gov) launch, the Target data breach, or really any number of multimillion-dollar projects that basically didn't work in the end. In 2012, the US Air Force scrapped an enterprise resource planning (ERP) software project after racking up \$1 billion in costs.

In cases like these, playing the blame game is customary. Even when most of those involved don't literally go down with the ship—as in the case of the *Titanic*—people get fired, careers get curtailed, and the Internet has a field day with both the individuals and the organizations.

But how do we square that with the frequent admonition to embrace failure in DevOps? If we should embrace failure, how can we punish it?

Not all failure is created equal. Understanding different types of failure and structuring the environment and processes to minimize the bad kinds is the key to success. The key is to “fail well,” as Megan McArdle writes in *The Up Side of Down: Why Failing Well Is the Key to Success* (Penguin Books, 2015).

In that book, McArdle describes the Marshmallow Challenge, an experiment originally concocted by Peter Skillman, the former VP of design at Palm.¹¹ In this challenge, groups receive 20 sticks of spaghetti, one yard of tape, one yard of string, and one marshmallow. Their objective is to build a structure that gets the marshmallow off the ground, as high as possible.

Skillman conducted his experiment with all sorts of participants from business school students to engineers to kindergarteners. The business school students did worst. I'm a former business school student, and this does not surprise me. According to Skillman, they spent too much time arguing about who was going to be the CEO of Spaghetti, Inc. The engineers did well, but also did not come out on top. As someone who also has an engineering degree and has participated in similar exercises, I suspect that they spent too much time arguing over the optimal structural design approach using a front-loaded waterfall software development methodology writ small.

By contrast, the kindergartners didn't sit around talking about the problem. They just started building to determine what works and what doesn't. And they did the best.

Setting up a system and environment that allows and encourages such experiments enables successful failure in Agile software development. It doesn't mean that no one is accountable for failures. In fact, it makes accountability easier because

¹¹<http://www.tomwujec.com/design-projects/marshmallow-challenge/>

"being accountable" needn't equate to "having caused some disaster." In this respect, it changes the nature of accountability.

We should consider four principles when we think about such a system: scope, approach, workflow, and incentives.

Scope

The right scope is about constraining the impact of failure and stopping the cascading of additional failures. This is central to encouraging experimentation because it minimizes the effect of a failure. (And, if you don't have failures, you're not experimenting.) In general, you want to decouple activities and decisions from each other. From a DevOps perspective, this means making deployments incremental, frequent, and routine events—in part by deploying small, autonomous, and bounded context services (such as microservices or similar patterns).

Approach

The right approach is about continuously experimenting, iterating, and improving. This gets back to the Toyota Production System's *kaizen* (continuous improvement) and other manufacturing antecedents. The most effective processes have continuous communication—think scrums and kanban—and allow for collaboration that can identify failures before they happen. At the same time, when failures do occur, the process allows for feedback to continuously improve and cultivate ongoing learning.

Workflow

The right workflow repeatedly automates for consistency and thereby reduces the number of failures attributable to inevitable casual mistakes like a mistyped command. This allows for a greater focus on design errors and other systematic causes of failure. In DevOps, much of this takes the form of a CI/CD workflow that uses monitoring, feedback loops, and automated test suites to catch failures as early in the process as possible.

Incentives

The right incentives align rewards and behavior with desirable outcomes. Incentives (such as advancement, money, recognition) need to reward trust, cooperation, and innovation. The key is that individuals have control over their own success. This

is probably a good place to point out that failure is not always a positive outcome. Especially when failure is the result of repeatedly not following established processes and design rules, actions still have consequences.

Culture

I said there were four principles. But actually there are five. A healthy culture is a prerequisite for both successful DevOps projects and successful open source projects and communities. In addition to being a source of innovative tooling, open source serves as a great model for the iterative development, open collaboration, and transparent communities that DevOps requires to succeed.

The right culture is, at least in part, about building organizations and systems that allow for failing well—and thereby make accountability within that framework a positive attribute rather than part of a blame game. This requires transparency. It also requires an understanding that even good decisions can have bad outcomes. A technology doesn't develop as expected. The market shifts. An architectural approach turns out not to scale. Stuff happens. Innovation is inherently risky. Cut your losses and move on, avoiding the sunk cost fallacy.

One of the key transformational elements is developing trust among developers, operations, IT management, and business owners through openness and accountability.

Ultimately, DevOps becomes most effective when its principles pervade an organization rather than being limited to developer and IT operations roles. This includes putting the incentives in place to encourage experimentation and (fast) failure, transparency in decision making, and reward systems that encourage trust and cooperation. The rich communication flows that characterize many distributed open source projects are likewise important to both DevOps initiatives and modern organizations more broadly.

Changing Culture

Shifting culture is always challenging and often needs to be an evolution. For example, Target CIO Mike McNamara noted in a 2017 interview that “What you come up against is: ‘My area can’t be agile because . . .’ It’s a natural resistance to change—and in some mission-critical areas, the concerns are warranted. So in those areas, we started developing releases in an agile manner but still released in a controlled environment.

As teams got more comfortable with the process and the tools that support continuous integration and continuous deployment, they just naturally started becoming more and more agile.”¹²

It’s tempting to say that getting the cultural aspects right is the main thing you have to nail in both open source projects and in DevOps. But that’s too narrow, really. Culture is a broader story in IT and elsewhere. For all we talk about technology, that is in some respects the easy part. It’s the people who are hard.

Writing in *The Open Organization Guide to IT Culture Change*, Red Hat CIO Mike Kelley observes how “This shift to open principles and practices creates an unprecedented challenge for IT leaders. As their teams become more inclusive and collaborative, leaders must shift their strategies and tactics to harness the energy this new style of work generates. They need to perfect their methods for drawing multiple parties into dialog and ensuring everyone feels heard. And they need to hone their abilities to connect the work their teams are doing to their organization’s values, aims, and goals—to make sure everyone in the department understands that they’re part of something bigger than themselves (and their individual egos).”¹³

Pervasive Open Source

Business models associated with open source software products are important to get right. It takes viable business models that involve, not just using, but contributing back to projects to sustain healthy open source communities. While many individuals are motivated to contribute to open source projects on their own time, the vast amount of open source software powering today’s world depends on corporations contributing as part of a profitable business plan.

Those viable business models exist today, notwithstanding the many challenges to getting them right and the temptation for companies to free ride or otherwise avoid contributing—topics that I’ll cover more deeply in the next chapter. In fact, many organizations have discovered broad benefits to participating in open source software development and even in adopting open source practices in other aspects of their business.

¹²<https://enterpriseproject.com/article/2017/1/target-cio-explains-how-devops-took-root-inside-retail-giant>

¹³<https://opensource.com/open-organization/resources/culture-change>

CHAPTER 6

The Flip Side of Open Source

This book has come from a place where open source software is a great success and transformational technology force. It would be hard to credibly argue that open source ideas and practices haven't greatly influenced how software gets developed and how cooperative innovation happens more broadly.

But even the most positive storylines embed other narratives that don't make it into press releases. Some of these are just blemishes. Others are more fundamental questions about balancing the desires of users and communities, creating sustainable development models, and continuing to flourish in an IT landscape that has fundamentally changed since open source first came onto the scene.

A Positive Feedback Loop

Perhaps the most glancing critique of the open source model is to point out that it's not always effective or that it's failed to make significant inroads in many areas.

Or, as Matt Asay has observed, that most open source projects rely on a small group of core contributors—often working for a vendor. The open source community development model is something of an idealization in many cases.¹

There's never going to be a singular approach to creating things that are useful to individuals and organizations. But it's worth asking what aspects of open source aren't always optimized.

¹<https://www.infoworld.com/article/3268001/open-source-tools/open-source-isnt-the-community-you-think-it-is.html>

The Linux Foundation's Jim Zemlin describes open source software development in terms of a positive feedback loop linking projects, solutions, and value. Maintaining the links in this feedback loop depends on there being answers to some important questions.

One is about successful business models; I've already touched on this question but will come back to it in depth later in this chapter; it's a defining question for open source success or failure. Other questions relate to aligning with communities, maintaining appropriate levels of control, and finding the right balance between sharing and consuming.

Projects

We've already covered many of the aspects of projects and their associated communities that it's important to get right. But, as Zemlin notes, there are always ways to improve.

"How do we create more secure code in upstream projects? How do we take the responsibility of this code being used in important systems that impact the privacy or the health of millions of individuals?" he asks. He adds that you may want to think explicitly about how your project fits into this feedback loop. "When I build my next project or a product, I should say, that project will be in line with, in a much more effective way, the products that I'm building."

Perhaps the biggest mindset shift over the past few years has been a broader recognition of the relationship of projects to value. As Zemlin puts it: "To get the value, it's not just consumed, it is to share back. There's not [just] some moral obligation, although I would argue that that's also important. There's an actual incredibly large business benefit to sharing as well." That said, it's hard to say free riding on the open source contributions of others is a solved concern for the open source model given how one-sided the participation of many organizations, including large IT vendors and cloud providers, can be.

Solutions and Products

Other questions relate to how products and other solutions work as part of an open source ecosystem. For example, there are many nuances involved in balancing competing requirements for stability and new technology across projects and commercial products. There is no single right answer here either. Tensions will always exist between upstream projects and downstream products—which in turn creates frictions that wouldn't exist in an idealized open source development model.

At the same time, commercial adoption and a community of support providers often need to move forward together and create a stable base upon which users and buyers are comfortable putting a project into production. Balancing the requirements for projects and products is hard. But finding a balance that works (whatever the compromises required) is necessary.

Successful products and other commercial solutions are often the necessary economic input into the open source feedback loop. Without that input, that is, customers paying for something they need, there may be no value created for those in a position to provide the inputs, such as developer salaries, which go into moving a project forward.

Value

It's a feedback loop and, as Zemlin argues, all three cogs are important. However, even if all aspects of open source as a development model and means for cooperation aren't solely about *business* value, that's an important point to probe if open source is going to continue to be one of the ultimate game changers.

Benefits such as societal good, better interoperability, and the creation of new companies are all well and good. But systematic success for open source depends on contributors to projects and their users realizing value—whether efficiency, speed of innovation, or reliability—which increases their opportunity to deliver profits. And for those contributors and users reinvesting patches, features, and resources into the project community, increasing the hiring of developers and others with expertise.

Value creation and reinvestment is central to the forces and market trends that could impinge on the viability of open source as an ongoing development model. These include macro changes in the IT industry, the shifting expectations and preferences of software users, and the effectiveness of business models that include open source software (or maybe even software more generally) as a central element.

The IT Industry Has Changed

Open source remains an instrument for preserving user options, portability, and sustainability.

When everything is proprietary, software can't really evolve in the same way to be a platform for other platforms. Look at the way Linux has evolved to be the foundation for new types of software and new types of open source platforms and technologies.

And this in turn has helped sustain the value of open source overall, which arguably comes most of all from its effectiveness as a software innovation accelerator.

How can open source software continue to sustain itself moving forward?

We live in a world increasingly distant in time from the Unix wars, which played a significant role in the genesis of open source. Of course, open source has continued to grow and thrive even though the importance of source code for supporting a fragmented collection of hardware platforms is mostly far behind us. Horizontal software stacks built on standardized hardware are the norm. But the IT industry of 2018 is a much different one from that of 2000 or even, really, 2010.

The Rise of the “Cloud”

Google’s then-CEO Eric Schmidt is often credited with coining the “cloud computing” term in 2006, although its first appearance was probably in a Compaq Computer business plan a decade earlier. As is so often the case with technology, though, closely related concepts had been germinating for decades. For example, in a 1961 speech given to celebrate MIT’s centennial, artificial intelligence pioneer John McCarthy introduced the idea of a computing utility.

As recounted by, among others, Nick Carr in his book *The Big Switch: Rewiring the World, from Edison to Google* (W.W.Norton & Company, 2013), the utility take on cloud computing metaphorically mirrored the evolution of power generation and distribution. Industrial Revolution factories in the late 19th century built largely customized, and decentralized, systems to run looms and other automated tools, powered by water or steam turbines.

These power generation and distribution systems, such as the machine works of Richard Hartmann in Figure 6-1, were a competitive differentiator; the more power you could produce, the more machines you could run, and the more goods you could manufacture for sale. The contrast to the electric grid and huge centralized power generation plants, such as the Hoover Dam shown in Figure 6-2, is stark.

Until recently, we’ve been living in an era in which the pendulum had clearly swung in favor of similarly distributed computing. Computers increasingly migrated from the “glass house” of IT out to the workgroups, small offices, and desktops on the periphery. Even before Intel and Microsoft became early catalysts for this trend’s growth, computers had been dispersing to some degree for much of the history of computing. The minicomputer and Unix revolutions were among the earlier waves headed in the same general direction.

You could think of these as analogs to the distributed power systems of the Industrial Revolution. And part and parcel of the environment that gave rise to open source. Where infrastructure is a competitive differentiator, the parts and knowledge to construct that infrastructure are valuable.

Today, however, that pendulum is swinging back. Amazon Web Services, Microsoft Azure, and Google Cloud Platform have emerged as the dominant global providers of cloud infrastructure services—which is to say pay-per-use building blocks to write your own applications. These, and many other, companies also offer consumers and businesses complete applications and other services. Gmail, Facebook, and Salesforce are among the most familiar examples.

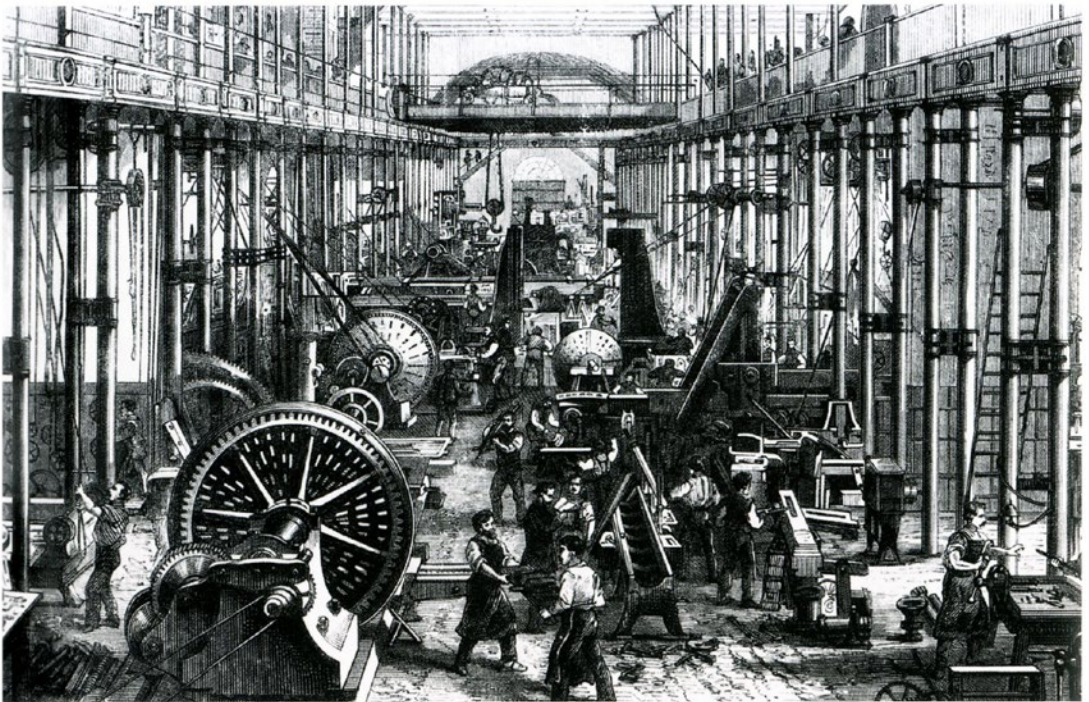


Figure 6-1. Machine works of Richard Hartmann in Chemnitz, Germany. The factories of the Industrial Revolution were built around localized and customized power sources that, in turn, drove local mechanical systems deriving from those power sources. Source: Wikimedia, in the public domain.



Figure 6-2. *The Hoover Dam (formerly Boulder Dam). Electric power as a utility has historically depended on large centralized power sources. Source: Across the Colorado River, 1942, by Ansel Adams. In the public domain from the National Archives.*

Why the Cloud Matters to Open Source

This shift of computing re-creates a new type of vertically integrated stack. One-time chief technology officer of Sun Microsystems, Greg Papadopoulos, one suspects hyperbolically and with an eye toward something IBM founder Thomas J. Watson probably never said, suggested that “the world only needs five computers,” which is to say there would be “more or less, five hyperscale, pan-global broadband computing services giants” each on the order of a Google.

Open source projects are ubiquitous throughout this new stack. Even Microsoft, now under CEO Satya Nadella, supports Linux workloads and has dropped the overt hostility to open source, which once characterized the company. (Former CEO Steve Ballmer once likened Linux to a “cancer” on the basis of its copyleft GPL license.)

However, remember the feedback loop. Some cloud giants have indeed made significant contributions to open source projects. For example, Google created Kubernetes, the leading open source project for managing software containers, based on the infrastructure it had built for its internal use. Facebook has open sourced both software and hardware projects.

But, for the most part, these dominant companies use open source to create what are largely proprietary platforms far more than they reinvest to perpetuate ongoing development in the commons. And they’re sufficiently large and well-resourced that they mostly don’t depend on cooperative invention at this point.

It’s easy to dismiss free riding as a problem given that organizations are missing out in some ways if they do so. However, to the degree that large tech companies, both cloud providers and others such as Apple, take far more from the open source commons than they contribute back, this at least raises concerns about open source sustainability.

Out at the Edge

It’s not wholly accurate to say that computing is recentralizing though. You probably have something in your pocket that has the capabilities of a supercomputer of not that many years ago. You may also have a talking assistant in your kitchen. Or a thermostat that’s connected to the Internet. Most computers these days aren’t in a datacenter.

However, all that distributed computing has important differences from the personal computer era of distributed computing.

Your iPhone is a walled garden that only lets you browse the Web or install Apple-approved applications on an Apple-developed software stack. The nature of the process effectively makes open source, at least in the sense of an iterative open development process, impossible.

Android phones run a variant of Linux, but you still don’t have the ability to modify them in the same manner as a PC. Sensors and other “edge” devices within the broad Internet-of-Things space can similarly run open source software but often depend on centralized services and, in any case, are usually designed as black boxes that resist user tinkering.

Thus, we increasingly find ourselves in a computing landscape composed of, on the one hand, monolithic computing services at the core and; on the other hand, mostly locked-down appliances on the edge. Neither is really a great recipe for the sort of cooperative infrastructure software development where open source has thrived.

Where's the Money?

The commercial value that can be extracted from software has also just declined more broadly. This might seem an odd assertion given how software and other technologies are indisputably central to more and more business processes and customer services. Stephen O'Grady explores this apparent contradiction in *The Software Paradox: The Rise and Fall of the Commercial Software Market* (O'Reilly Media, 2015). He argues that "Software, once an enabler rather than a product, is headed back in that direction. There are and will continue to be large software licensing revenue streams available, but traditional high margin, paid upfront pricing will become less dominant by the year, gradually giving way to alternative models."

In other words, software enables organizations to extract value from other things that they sell. Apple has been so successful, in part, because it sells a complete integrated product of which the operating system is an incidental part. And, indeed, app developers for iOS find it a challenging business because offering apps solely through up-front purchases mostly doesn't work at all.

Open source models arguably exacerbate the issue. Stephen O'Grady has also written that "The numbers, on the surface, would indicate that the various economic models embraced by open source firms are not in fact favorable to the firms that embrace them. Closed source vendors are typically appalled at the conversion rates of users/customers at firms like JBoss (3%) and MySQL (around 1%, anecdotally, on higher volume)—and those firms are more or less the most popular in their respective product categories. Even with the understanding that these are volume rather than margin plays, there are many within the financial community that remain skeptical of the long term prospects for open source (even as VC's pour money in in hopes of short term returns)."²

He wrote those words over 10 years ago, but Red Hat's success notwithstanding, there remains a dearth of companies that have been able to turn pure open source plays into significant revenues and profits. Red Hat has done many things right in terms of

²<http://redmonk.com/sograpy/2006/07/31/billion-dollar-open-source-businesses/>

both strategy and executing. Analyst Krish Subramanian attributes it “to 1) Being the first to understand OSS model and establish themselves well before the industry woke up to OSS 2) Picking the right OSS projects and *contributing code* to these projects. You can’t win in OSS without code contribution.”

Over that period, the industry has probably also developed a better appreciation for some approaches to business using open source that don’t work. Developer tools have always been a tough business, even with expensive proprietary products. Selling open source subscriptions for client devices is also hard; lack of good business models aren’t the only reason that we’ve never seen “The Year of the Linux Desktop” but it hasn’t helped. We see similar dynamics with the Internet-of-Things. The webcams, temperature sensors, and smoke alarms may be running open source software but essentially no one is willing to pay for that in the consumer market especially.

In any case, building a significant business around selling open source software is clearly not straightforward or a lot more companies would have done so. It’s also worth observing that even Red Hat, at about \$3 billion dollars in annual revenues as of 2018, is still quite small compared to either traditional proprietary software vendors or the new cloud vendors who depend upon open source. Amazon Web Services alone has over \$20 billion in (rapidly growing) annual revenue.

What Users Want

User needs have also changed. In some respects, this applies more to individual consumers than to corporate buyers. But, as IT consumerizes, the boundaries between those groups blur and fade to the point where it can sometimes make sense to think about users broadly.

The New Bundles

Bundling is a broad concept and there’s perhaps no more better historical example than newspapers.

Newspapers bundle various news topics like syndicated and local news, sports, and political reporting, along with advertising, classifieds, weather, comic strips, shopping coupons, and more. Many of the economic woes of the newspaper can be traced to the splitting of this bundle. Craigslist took over the classifieds—and made them mostly free.

Online severed the connection between news and local ads. While ads run online as well, the economics are something along the lines of print dollars devalued to digital dimes.

As NYU professor Clay Shirky wrote in 2008: “For a long time, longer than anyone in the newspaper business has been alive in fact, print journalism has been intertwined with these economics. The expense of printing created an environment where Wal-Mart was willing to subsidize the Baghdad bureau. This wasn’t because of any deep link between advertising and reporting, nor was it about any real desire on the part of Wal-Mart to have their marketing budget go to international correspondents. It was just an accident. Advertisers had little choice other than to have their money used that way, since they didn’t really have any other vehicle for display ads.”³

Software, both historically and today, has also been a bundle in various respects. As O’Grady observed in *The Software Paradox*, software used to be mostly something that you wrote in order to be able to sell hardware. It wasn’t something considered valuable on its own. In many respects, we’ve increasingly returned to that way of thinking whether we’re talking iOS on an iPhone or Linux on a webcam.

One of the core challenges around business models built directly on open source software is that open source breaks software business models that depend on bundling. If all you care about is the code in the upstream project, that’s available to download for free. An open source subscription model doesn’t require you to keep paying in order to retain access to an application as a proprietary subscription does. You’ll lose access to professional support and other benefits of the subscription, but the software itself is gratis. In fact, open source is arguably the ultimate force for unbundling. Mix and match. Modify. Tinker. Move.

Online consumer services in particular have also just led to a mindset that you don’t have to directly pay for many things. Your data is being mined and you’re being targeted by advertising, but your credit card isn’t being billed monthly. Or, you’re buying something else from the company in question and the software you use is simply in support of that. There’s a widely held expectation that, if it’s digital, it should be free. This echoes *The Software Paradox* again.

³https://www.edge.org/conversation/clay_shirky-newspapers-and-thinking-the-unthinkable

Users Want Convenience

There's another aspect of bundles worth exploring.

Bundles, like other aspects of packaging, are prescriptive. They can be seen as a response to *The Paradox of Choice: Why More Is Less* (HarperCollins), a 2004 book by American psychologist Barry Schwartz, in which he argues that consumers don't seem to be benefiting psychologically from all their autonomy and freedom of choice. Whether or not one accepts Schwartz's disputed hypothesis, it's certainly the case that technology options sometimes seem to proliferate endlessly with less and less real benefit to choosing one tech over another.

When sellers create bundles they also do so for a variety of reasons that often have to do with getting people to pay for stuff that they don't really want to pay for. The auto manufacturer who will only install leather seats if you also buy a sunroof and an upgraded trim package isn't doing so primarily because it wants to make life easier for the buyer. It's doing so because not everyone who *really* wants leather seats would normally be willing to also pay for those other options given the choice.

But the experience and convenience of a well-designed bundle that extends beyond the core product is at least a side effect in many cases.

Unbox a computer a couple of decades ago and, if you were lucky, you might find a sheet of paper easily identifiable as a "Quick Start" guide. (Which itself was an improvement over simply needing a field engineer to swing by.)

Today the unboxing experience of consumer goods like Apple's iPhone has become almost a cliché, but it's no less real for that. In the words of Grant Wenzlau, "Packaging is no longer simply about packaging the object—it is about the unboxing experience and art directing. This is where the process starts for designers today: you work backward from the Instagram image to the unboxing moment to the design that serves it."⁴

Stephen O'Grady also writes about the power of convenience that bundled services can help deliver. "One of the biggest challenges for vendors built around traditional procurement patterns is their tendency to undervalue convenience. Developers, in general, respond to very different incentives than do their executive purchasing counterparts. Where organizational buyers tend to be less price sensitive and more focused on issues relating to reliability and manageability, as one example, individual

⁴<http://www.thedieline.com/blog/2016/1/13/emerging-packaging-design-trends-of-2016-essentialism>

developers tend to be more concerned with cost and availability—convenience, in other words.”⁵

He argues that, while Napster became popular in part because people could download music for free, it was also more convenient than driving to the mall and buying an album on a CD. Today, streaming accounts for the majority of music industry revenue with over 30 million subscribers in the United States.

Open source software can still have a good user experience, of course. And some communities provide bundling such as Fedora for Linux and the OpenShift Origin for container platforms. However, these are (at least in part) best efforts by the community and—though reasonably stable and satisfactory for some use cases—there is no support model that ensures that parts work well together or that integration problems will be fixed promptly as there is in the commercial products deriving from these projects.

Using open source software doesn’t need to be about self-supporting a bag of parts downloaded off some repository on the Internet. That’s what commercial software subscriptions are for: turning unsupported projects into supported and curated products. Nonetheless, there’s at least a tension between the idea of open source software as malleable, customizable, and free (in either meaning of the word) and a bundle that, by design, is prescriptive, abstracts away underlying complexity, and explicitly excludes technologies not deemed sufficiently mature for the target buyer.

Hiding Complexity

The history of the computer industry has been one of continuously layering abstractions. The operating system was one of the first. Others have included virtual memory, logical addressing in disks, and virtualization in its many forms.

This can lead to abstracting away the infrastructure components where active open source communities have historically created so much software value.

We see an example in one of the most recent software trends, nascent as of this writing, is what usually goes by the monikers “serverless computing” and Functions-as-a-Service. (There’s still a server, of course, but the idea is to make it invisible to developers.) This abstracts away underlying infrastructure to an even greater degree than containers, allowing for suitable functions—think encoding an uploaded video file—to run in response to events and other triggers. As with many technologies, the

⁵<http://redmonk.com/sogady/2012/12/19/convenience/>

general concept isn't exactly a new one. For example, IBM CICS, invented 50 years ago, provides services that extend or replace the built-in functions of the operating system.

Most associated with Lambda at Amazon Web Services currently, a variety of open source projects in this space such as OpenWhisk and Knative are also underway. The overall goal can be thought of as almost making the packaging invisible while letting developers implement an idea with as little friction as possible.

The idea is a sound one. In some respects, serverless extends the container concept to further mask complexities that aren't relevant to many developers' writing applications. However, at the same time, it at least raises the question of whether ongoing community investments in platform software will decline as the value and attention shifts to code that's specific to individual businesses.

It's Not the End

If this chapter so far has been a bit of a downer, it's because it's important to poke at the open source model as a way to better understand strengths and weaknesses and to guide future actions. To that end, consider this hand-wringing in the context of both the IT industry and the broader market forces at play.

IT Is Hybrid

One context is simply the observation that the adoption of technology takes place over a period of years and even decades in different places and at different rates. Science fiction author William Gibson once said that "The future is already here—it's just not very evenly distributed."

That pretty much describes technology adoption patterns. The Silicon Valley startup all-in on Amazon Web Services may have little in common with the regional bank that's still getting its virtualization workflows in order—much less the local pizza shop running an ancient version of Windows on its point of sale system.

The Big Switch metaphor for cloud computing, by which electric power centralized and became a commodity from the perspective of users, arguably became a staple of many a conference presentation because of such absolutes. The advantages of centrally generated electricity over complex arrays of belts and pulleys that dictated how factories

needed to be arranged, however inefficient from a workflow perspective, were profound. Electricity could be sent through wires to motors attached to individual pieces of machinery. The thousands of motors in a modern manufacturing plant range from those in huge gantry cranes to numerous small ones in power tools.

As a result, although the changeover was slow at first because of the initial expense and the novelty of electric power, factories began to switch to electricity in earnest during the first decade of the 20th century. In *The Big Switch: Rewiring the World, from Edison to Google*, Nick Carr writes that “By 1905, a writer for Engineering magazine felt comfortable declaring that ‘no one would now think of planning a new plant with other than electric driving.’ In short order, electric power had gone from exotic to commonplace.”

It’s something of an irony that, while cloud computing was coming onto the scene, electricity generation was actually starting to *decentralize* with the increased use of solar panels in particular. But the more important point is that computing as a new utility on the order of the electric grid is mostly wrong—at least for any time values that we care about as a practical manner.

Consider all the reasons businesses, especially large ones, might want or need to continue to run applications in-house: control and visibility, compliance with regulations, integration with various existing software and hardware, proximity to data, and so forth. In short, we should expect a future that is a hybrid of many things, not just one big switch. This doesn’t make for a tidy narrative in which some specific approach or technology conquers all. But it’s far more consistent with the observed history of the computer industry in which, as we’ve seen, there may be certain overarching ebbs and flows, but messiness generally wins out over out over winner takes all.

Stipulate, if you will, both that public clouds will increase in their share of the computing market and that the major global public clouds, as they operate today, don’t necessarily have incentives to significantly invest in open source. Even if that’s the case, many players in the IT industry will continue to have those incentives, including smaller cloud and other service providers who can take advantage of open source projects like OpenStack to compete on a more equal footing.

Centers of Gravity Shift

Where has open source software development had the greatest impact? The easy response is the operating system—notably Linux. You wouldn’t necessarily be wrong. Linux brought a Unix-like operating system to mass market hardware and

unified the non-Microsoft Windows computing world to a degree that would almost certainly never have happened otherwise. It's big in mobile, by way of Android, and it is relentlessly filling other niches that were once reserved for proprietary operating systems.

A more complete response would take a broader view that included software such as the Apache web server, innovations in software-defined storage and software-defined networking, OpenStack, container platforms, and more. However, whether we take the narrow view or the broader one, it's true that the focus of open source development has been on infrastructure software, the software plumbing of datacenters that's largely independent of an organization's industry or specific application requirements.

It's then a short step to argue that at some of this base plumbing is very well-understood and mature. It may not exactly be a commodity; for example, security incidents highlight the need for ongoing operating system support and updates. But it's hard to argue with a general assertion that, as we've seen, software value is shifting to applications that businesses use to create differentiated product and service offerings. Furthermore, computing infrastructure is precisely what cloud providers offer. It's not necessary to install and operate it yourself.

This is all true. However, open source software development already has a long history of moving into new areas. In part, it's about moving up the stack from the operating system layer to software that builds on and expands on the operating system, such as container orchestration. Or, into expanding middleware for both traditional enterprise applications and new areas such as the Internet-of-Things.

The Development Model Is Effective

And, if the open source development model is effective—as it certainly appears to be in many cases—why wouldn't we expect to see its continued use? True, we can point to examples of very successful companies that extensively use open source software but don't participate much in the open development process and others that are simply mostly proprietary. However, as we've seen, many companies—including many who aren't traditional IT vendors—do participate in various aspects of open source development because it makes business sense for them to do so.

It's possible, perhaps even probable, that the focus of open source development will broaden and shift over time. For example, with Automotive Grade Linux, we see specialized development that builds on a horizontal platform to better meet the needs of a particular industry. We see a corresponding involvement by companies that we would have historically categorized as end-users in Industrial Internet-of-Things software projects.

There's no shortage of software needed across many industries and a relatively limited population of skilled developers to write it. The landscape won't stay static. Different organizations will participate in different areas. Who profits directly and indirectly from open source will change.

But even with the changed and changing landscape in computing, there really aren't good reasons to think that open source as a software development model is something that only made sense for a relatively fleeting window of time.

Ecosystems Matter

It's also worth remembering where some of the impetus behind open source comes from.

Vertical integration was a common model for 20th-century industrial companies. As *The Economist* notes, "Some of the best known examples of vertical integration have been in the oil industry. In the 1970s and 1980s, many companies that were primarily engaged in exploration and the extraction of crude petroleum decided to acquire downstream refineries and distribution networks. Companies such as Shell and BP came to control every step involved in bringing a drop of oil from its North Sea or Alaskan origins to a vehicle's fuel tank."⁶ In its heyday, Eastman Kodak owned its own chemical company to meet its needs for the vast quantities of ingredients needed to manufacture and process film.

We've already seen how commonplace this model was in the computer industry with mainframes; minicomputers; and, to an only somewhat lesser degree, Unix systems.

It could be an effective way to control access to inputs. It could also increase the market power of a dominant supplier. Indeed, regulators and lawmakers have at times restricted certain forms of vertical integrations as well as other types of tying together

⁶<https://www.economist.com/node/13396061>

products and services. For example, manufacturers have often tried to make using hardware like a printer contingent on buying a profitable stream of printer supplies like ink from them, rather than going to a discount third party.

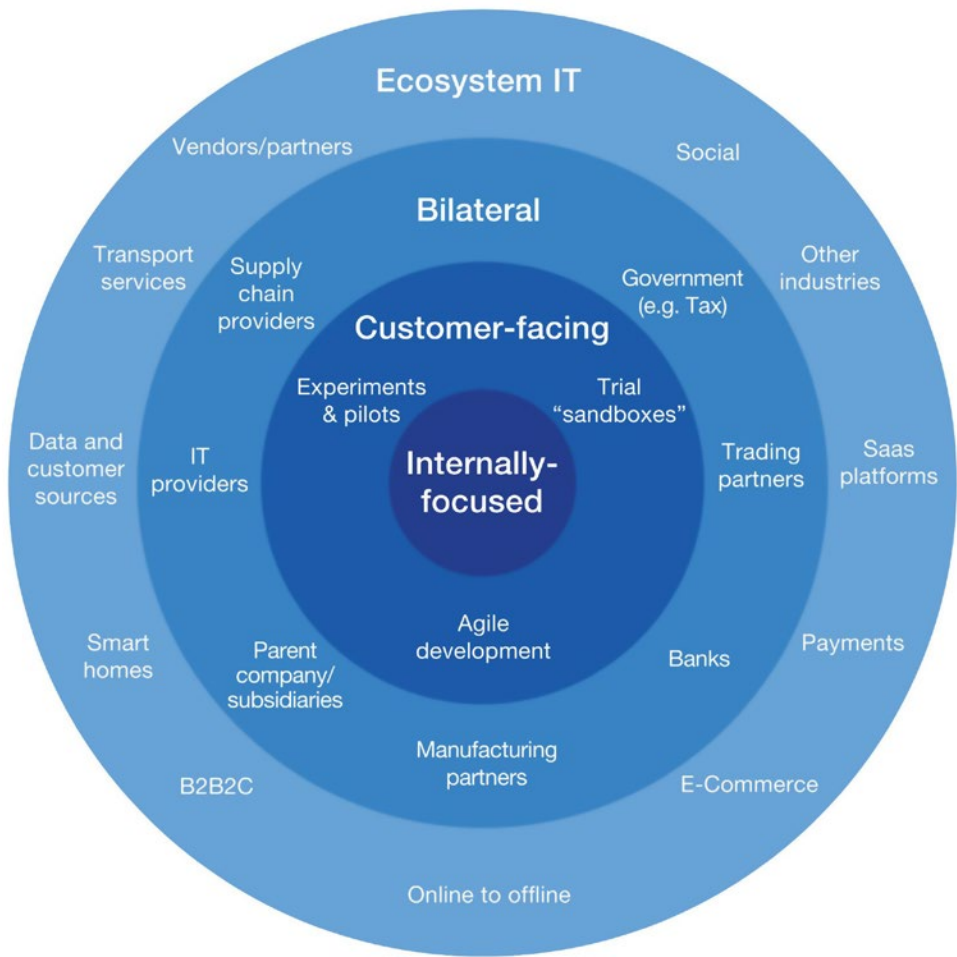
Today though we see the rise of coopetition. We see markets demanding interoperability and standards. We see more specialization and disaggregation.

Furthermore, ecosystems are only expanding. Management consultants McKinsey refers to “competing in a world of sectors without borders.”⁷ They go on to write that “We’ve all experienced businesses that once seemed disconnected fitting together seamlessly and unleashing surprising synergies: look no farther than the phone in your pocket, your music and video in the cloud, the smart watch on your wrist, and the TV in your living room.” In another report, McKinsey talks of a “radical reframing of what IT is and how CIOs manage it—not as an internal collection of information technologies (IT) but as a broad network of ecosystem technologies (ET).”⁸ Their “four-layered IT” model based on this perspective is illustrated in Figure 6-3.

⁷<https://www.mckinsey.com/business-functions/mckinsey-analytics/our-insights/competing-in-a-world-of-sectors-without-borders>

⁸<https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/adopting-an-ecosystem-view-of-business-technology>

Overview of four-layered IT



McKinsey&Company

Figure 6-3. McKinsey argues that to fully benefit from new business technology, CIOs need to adapt to emerging technology ecosystems. Source: Adopting an ecosystem view of business technology, February 2017.

Together, these suggest an environment that naturally aligns with organizations working together on technologies and other projects for mutually beneficial reasons. Open source software isn't the only way to do so. But much of today's technology is software. Standards are increasingly developed first through code. Open source development can almost be thought of as a communication platform for disparate companies working together to achieve business objectives.

It's Not Just About the Code

I've laid out some of the challenges that open source faces in today's world. IT organizations can increasingly use global cloud providers rather than operate their own infrastructure. That same infrastructure is being commoditized in many cases and community code, downloaded for free, is good enough for many purposes. This in turn can choke off the reinvestment that the open source development model needs to work.

Furthermore, the viability of software as a stand-alone business seems to be declining. Users increasingly don't want to explicitly pay out of pocket for software. They expect it as part of a bundle whether that means hardware or paying implicitly through advertising and other forms of monetization, which makes their attention something of a product to be sold.

And users want convenience. Open source was born from an era where using software meant installing it on a computer someplace. Today, they're more likely to acquire some sort of service that runs on software, invisible to them. Software abstracted in this way can be less valuable for a company looking to profit from it and reinvest some of those proceeds in its ongoing development.

At the same time, it's not all doom and gloom. Open source is a good fit with developing ecosystems and businesses that need to work together on technology.

However, there's a bigger point, which is the topic of this book's final chapter. Open source principles have indeed led to an extremely effective software development model that can form the basis for equally effective enterprise software products. Products that bring not just code, but also expertise, support, and other capabilities to solve business problems, to customers. That model has also started to influence the way that software is developed more broadly.

It also goes beyond software. Many principles, or at least analogs to them can influence approaches to data, to education, to organizations. There's more inertia in many of these areas than in software. But there are also enormous opportunities.

CHAPTER 7

Open Source Opportunities and Challenges

The prior chapters of this book focused on “open” primarily as it relates to software code and the software development model.

As we have seen, opening up code solved practical problems facing a computer industry with a fractured hardware and software landscape. Later, the open source development *process* became an increasingly important component of open source. In this case, the practical problems involved improving cooperation, innovation, and interoperability.

However, many of the same principles and practices—or at least echoes of them—can apply to other areas. And, indeed, we see that happening. This chapter explores openness as it extends beyond source code.

In some cases, it involves sharing of and collaboration around other forms of digital artifacts, whether raw data or other forms of information. Interactive communication and development of knowledge is also part of the education and research process. Open source principles can be applied to hardware. Finally, there are the big questions of structure and the way organizations work. Do open source communities have anything to teach us?

Opening Data

Determining what it means for data to be open, why we might want (or not want) data to be public, how datasets interact, and the practical difficulties of moving data have a lot of different angles. But considering value, transparency, and ownership hits most of the high points.

Value from Data

Data and computer programs have a close relationship. A program often takes in one set of data—for example, a grid of temperature, humidity, and atmospheric measurements over a region of ocean—and transforms it into another set of data. Perhaps a storm track or intensity prediction. Absent data—and, in this case, data that’s not too old—the most powerful supercomputer and sophisticated weather model is worthless.

Data has long had value of course, but it was in the mid- to late 2000s that a widespread appreciation of that fact began to really sink in.

In 2006, Clive Humby, a UK mathematician and architect of Tesco’s Clubcard, coined the since widely quoted phrase: “Data is the new oil.” Less quoted is his follow-up, which emphasized that raw data has to be processed and analyzed to be useful. “It’s valuable, but if unrefined it cannot really be used. It has to be changed into gas, plastic, chemicals, etc to create a valuable entity that drives profitable activity; so must data be broken down, analyzed for it to have value,” he said.

However, in 2008, *Wired* magazine’s Chris Anderson wrote an article titled “The End of Theory: The Data Deluge Makes the Scientific Method Obsolete.” His thesis was that we have historically relied on models in large part because we had no other choice. However, “with enough data, the numbers speak for themselves.”¹

It was a provocative point at the time and it remains a likely deliberate overstatement even today. However, the intervening decade has seen dramatic increases in machine learning performance across a large number of areas. For example, in certain types of tests, computers can actually outperform humans at image recognition today.

While the techniques to use data effectively continue to be refined, many of the most impressive “AI” achievements draw heavily from work that Geoff Hinton did back in the 1980s on a generalized back-propagation algorithm for training multilayer neural networks. These advances have come about partly through increases in compute power, especially in the use of graphics and specialized processors like Google’s Tensor Processing Units (TPU). They also depend on huge datasets used to train and validate the machine learning models.

It’s not as simple as saying “it’s all about the data” but there’s clearly been a shift in value toward data. As a result, companies like Google and Facebook are far more amenable to participating in open sourcing code than they are at opening their data (and the specific ways they work with that data such as search algorithms).

¹http://www.wired.com/science/discoveries/magazine/16-07/pb_theory

One example of open source principles and practices being applied to data is OpenStreetMap.

Steve Coast founded the project in 2004, which was initially focused on mapping the United Kingdom, where Ordnance Survey mapping data wasn't freely available. (By contrast, in the United States, map services and data downloaded from The National Map maintained by the US Geological Survey (USGS) are free and in the public domain.) In April 2006, the OpenStreetMap Foundation was established to encourage the growth, development, and distribution of free geospatial data and provide geospatial data for anybody to use and share. OpenStreetMap is a particularly good example of open data because the data generated by the OpenStreetMap project is considered its primary output rather than map tiles or other services that can be created from the data.

According to the project, "OpenStreetMap emphasizes local knowledge. Contributors use aerial imagery, GPS devices, and low-tech field maps to verify that OSM is accurate and up to date." It also allows for automated imports of data from sources that use appropriate licenses.

The resulting data is under the Open Database License. It is free to use for any purpose so long as OpenStreetMap and its contributors are credited.

Comparing the quality of OpenStreetMap with commercial services such as Google Maps is difficult because quality tends to be a function of both the location and which features are important to you. For example, OpenStreetMaps lacks the real-time information about road and traffic conditions needed to do effective routing. On the other hand, commercial services often draw data from sources that don't emphasize features such as hiking trails which are more likely to be represented on OSM.

In general, though, it's fair to say that both the quantity and quality of data in OpenStreetMap has improved dramatically over the past decade or so, and it's a valuable open data resource for many purposes.

Transparency Through Data

Data can be valuable in the monetary sense. It enables services that are useful and for which consumers and organizations are willing to pay. As machine learning seems poised to be an integral part of more and more technology from automobiles to conversational assistants like Amazon's Echo, data seems certain to only become more valuable—leading to increased tensions between opening data and profiting from it.

However, another aspect of data openness is the increased availability of data from public institutions and others so that citizens can gain insight into government actions and the environment they live in.

There are nuances to open data in this context.

Not all data collected by a government is or should be publicly available. The specifics will differ by country and culture—for example, salary information is more public in some countries than in others—but individuals and institutions will always (properly) have their secrets.

Furthermore, when data is opened, it needs to be done in a way that it's actually useful. Red Hat's Melanie Chernoff writes that "What we, as policy advocates, want to encourage is that the data that governments do and should publish is done so in a way to ensure equal public access by all citizens. In other words, you shouldn't have to buy a particular vendor's product in order to be able to open, use, or repurpose the data. You, as a taxpayer, have already paid for the collection of the data. You shouldn't have to pay an additional fee to open it."²

It's easy to find fault with governmental and corporate transparency generally. However, the overall trajectory of making public data available is mostly positive.

For example, in May of 2013, then US president Obama signed an executive order that made open and machine-readable data the new default for government information. Over a quarter of a million datasets are currently available.³

Many municipalities also make available various types of data available about crime, traffic infractions, health code violations, zoning, and more.

How is this data used?

A common technique is to "mash up" data with maps. Humans are visual creatures, so presenting temperatures, crime statistics, or population densities on a map often makes quickly discerning patterns and spatial relationships easier than presenting the same facts as a boring table.

Some uses are mostly tactical and practical. USGS river data level is used by whitewater paddlers and others to plan their trips. National Weather Service, or specifically the forecasts based on it, help you decide whether to pack an umbrella in the morning.

²<https://opensource.com/government/10/12/what-%22open-data%22-means-%E2%80%93-and-what-it-doesn%E2%80%99t>

³<https://www.data.gov/>

However, other types of data can give insight into the “health” of cities or actions taken by public employees. One example from New York City revealed that police officers were routinely ticketing cars in locations where it was, in fact, legal to park under an amended law.⁴ Other examples include heat maps of where different types of crime occur. Individuals and companies can also build on public data to create applications that make it easier to use mass transit effectively or to more easily find out whether a restaurant has been cited for health concerns.

A common theme is that combining a multiplicity of datasets can often yield nonobvious insights.

Ownership of Data

The flip side to transparency and one that is much on the minds of regulators and others as of 2018 is the ownership of data and privacy issues associated with it.

Eben Moglen of the Software Freedom Law Center once referred to Google and its ilk as a “private surveillance system that [the government] can subpoena at will.” He went on to describe this as a “uniquely serious problem.” It’s hard to dispute that such services create an unprecedented centralization of data—both for data explicitly placed into their cloud and that generated with each search or purchase. This is anathema to those who saw open source and inexpensive computers as a great victory for the decentralization of computing and information.

The monetization of essentially private behaviors like browsing the Web for advertising and marketing purposes is part of the impetus behind regulations such as the European Union’s GDPR. The widespread availability and archiving of so much data have also exposed some of the fault lines between overall US and European attitudes toward free press, free speech, and privacy.

At a higher level, societies as a whole have also not really come to grips with what it means for so much data about individuals to be so widely available and combinable. It’s easy to focus on the usual suspects. Facebook. Equifax. Google. They’re profiting from mining our online behaviors and selling the results to advertisers and marketers. They make convenient bogeymen. (Perhaps some of their behaviors should even be curtailed.)

⁴<http://iquantny.tumblr.com/post/144197004989/the-nypd-was-systematically-ticketing-legally>

But there are broader questions. As we've seen, sharing data can be a positive. Speaking at the 2018 MIT Sloan CIO Symposium in May, Elisabeth Reynolds, the executive director of the Work of the Future Task Force, observed that "the regulatory framework is often opposed to sharing data for the greater good."

For example, data about vehicle traffic, electricity usage, or personal health metrics could potentially be aggregated and studied to reduce congestion, optimize power generation, or better understand the effectiveness of medical treatments.

However, the more fine-grained and comprehensive the data set, the harder it is to truly anonymize. The difficulty of effective anonymization is well-known. Either your search history or your car's GPS tracks would likely leave little doubt about who you are or where you live.

The problem is only compounded as you start mixing together data from different sources. Gaining insights from multiple data streams, including public data sets, is one of the promises of both IoT and AI. Yet, the same unexpected discoveries made possible by swirling enough data together also make it hard to truly mask identities.

Furthermore, there's personal data that's public for sound policy reasons, often to do with transparency. The sale of your house and your deed are public documents in the United States. But there's a difference of degree that's sufficient to become a difference of kind when all those public records are a mouse click away and ready to be combined with countless other datasets rather than filed away deep in some dusty county clerk's office.

Openness of data is mostly a good thing. But it can be hard to strike the appropriate balance. Or even to agree what the appropriate balance should be.

Opening Information

The first phase of the Web, including the dot-com bubble, majored in static websites and e-commerce. There was an increasing amount of information online but the typical web surfer wasn't creating much of it.

As the industry recovered from the burst dot-com bubble, that began to change.

The Read/Write Web

One aspect of this shift was "Web 2.0," a term coined by O'Reilly Media to describe a web that was increasingly characterized by connections, which is to say a form of data describing relationships. In Tim O'Reilly's words, "Web 2.0 doesn't have a hard

boundary, but rather, a gravitational core. You can visualize Web 2.0 as a set of principles and practices that tie together a veritable solar system of sites that demonstrate some or all of those principles, at a varying distance from that core.”⁵

Another way to look at this phase comes from the inventor of the World Wide Web, Tim Berners-Lee, who used the term “read/write” web to describe a web in which everyone created as well as consumed.⁶ This was around the time that blogging was really taking off and content created by users became an important part of the data and information underpinning the web.

Besides the Web as a whole, the best example of a large-scale collaboratively created open information resource is probably Wikipedia.

Wikipedia

Launched in 2001 by Jimmy Wales and Larry Sanger, Wikipedia wasn’t the first (or last) attempt to create a crowdsourced encyclopedia. In a 2011 article, Megan Garber, writing for NiemanLab, counted six prior attempts. Subsequent attempts at different takes on the Wikipedia model, such as Google’s Knol, were likewise unsuccessful.

The reasons for Wikipedia’s success have never been completely clear. In 2011, research by Berkman fellow and MIT Media Lab/Sloan School of Management researcher Benjamin Mako Hill⁷ suggested it attracted contributors because it was a familiar product (an encyclopedia), focused on content rather than technology, offered low transaction costs to participation, and de-emphasized individual ownership of content.

One also suspects that it stumbled into a governance model that struck at least a workable balance between autocratic control structures and a crowdsourced free for all. There’s no shortage of complaints, even today, about various aspects of how Wikipedia is administered and certainly quality can be uneven. However, it’s hard to dispute the fact that Wikipedia mostly works and is a valuable resource.

⁵<https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html?>

⁶<http://news.bbc.co.uk/2/hi/technology/4132752.stm>

⁷<http://www.niemanlab.org/2011/10/the-contribution-conundrum-why-did-wikipedia-succeed-while-other-encyclopedias-failed/>

Independent Contribution

Other sites are open and collaborative in the sense that they provide a platform for many users to contribute content, under a Creative Commons open source license or otherwise. YouTube is well-known for videos. Flickr, recently purchased by SmugMug, is one example for photos although—in terms of sheer numbers—more social media-oriented properties like Instagram (owned by Facebook) are much larger.

The differences in the collaborative approach between Wikipedia and these other sites are probably rooted in the type of content. Collaboratively edited encyclopedia articles may suffer a bit from not having the coherence that a single author or editor can bring. But diverse perspectives that also guard against bias and self-promotion are usually a net win. On the other hand, collaboration on a posted photograph or video doesn't really make sense in most cases other than remixing or incorporating into new works.

Another large digital library is The Internet Archive, founded by Brewster Kahle in 1996, which has a stated mission of “universal access to all knowledge,” including nearly three million public domain books. The Internet Archive allows the public to upload and download digital material, but the bulk of its data is collected automatically by its web crawlers, which work to preserve copies of the public web except where owners explicitly request otherwise. This service, which allows archives of the Web to be searched and accessed, is called The Wayback Machine, a play on the time machine in “Peabody's Improbable History,” a recurring feature of the 1960s cartoon series *The Rocky and Bullwinkle Show*.

Certainly, vast quantities of information remain behind paywalls and firewalls—or locked up on paper, microfilm, or microfiche—but the amount of information that is readily available with limited restrictions on its use would still be almost unfathomable just a few decades ago.

Opening Education

Probably the simplest (if incomplete) definitions of open education focus on reducing barriers to education, whether admission requirements, cost, or other factors.

Of course, teaching and learning took place long before there were formal educational institutions. Our distant ancestors didn't sign up for courses in wooly mammoth hunting in order to learn how to feed themselves.

Precursors

In the United States, an early example of what starts to look like an open education program comes from 4-H clubs. (4-H derives from the organization’s original motto: head, heart, hands, and health.) It grew out of a desire to expose children to practical and “hands-on” learning that connected public school education to country life—and, in turn, expose their parents to the new agricultural approaches and technologies they tended to resist. According to the organization, “A. B. Graham started a youth program in Clark County, Ohio, in 1902, which is considered the birth of 4-H in the United States. The first club was called ‘The Tomato Club’ or the ‘Corn Growing Club.’ T.A. Erickson of Douglas County, Minnesota, started local agricultural after-school clubs and fairs that same year. Jessie Field Shambaugh developed the clover pin with an H on each leaf in 1910, and by 1912 they were called 4-H clubs.”

4-H and related programs persist in many rural and semi-rural communities today as seen in the many country fairs that are a summer and autumn staple in parts of the United States such as the town where I live.

MIT OpenCourseWare

Fast forward to the Internet age and hit pause at 2001. That was the year that the Massachusetts Institute of Technology first announced MIT OpenCourseWare, which is an obvious point at which to start our look at where open education stands today. It was formally launched about two years later.

The concept grew out of the MIT Council on Education Technology, which was charged by MIT provost Robert Brown in 1999 with determining how MIT should position itself in an environment where other schools were increasingly offering distance learning options to paying students.

MIT took a rather different approach. For one thing, the content was made available for free under an open license.

As Carey Goldberg of *The New York Times* reported, Steven Lerman, the faculty chairman, argued that “Selling content for profit, or trying in some ways to commercialize one of the core intellectual activities of the university seemed less attractive to people at a deep level than finding ways to disseminate it as broadly as possible.”⁸

⁸<https://www.nytimes.com/2001/04/04/us/auditing-classes-at-mit-on-the-web-and-free.html>

For another, the emphasis was on providing the raw ingredients for educators rather than an education in and of itself. In announcing the initiative, MIT President Charles Vest said that “We are not providing an MIT education on the Web. We are providing our core materials that are the infrastructure that undergirds an MIT education. Real education requires interaction, the interaction that is part of American teaching. We think that OpenCourseWare will make it possible for faculty here and elsewhere to concentrate even more on the actual process of teaching, on the interactions between faculty and students that are the real core of learning.”

Many other schools announced programs in the same vein over the next few years and, in general, these commons of educational resources have grown over time.

We’ll return to this idea of open educational resources (OER) shortly but, first, no discussion of open education would be complete without mentioning massive open online courses (MOOCs).

MOOCs

As Vest noted, the commons populated by MIT OpenCourseWare and other resources of its type were intended as resources for learning rather than a course in a box. In addition to the reasons Vest gave, one suspects that the additional distance this approach provided between in-person MIT courses and MIT OpenCourseWare ones made the concept an easier sell. At a practical level, ubiquitous video and audio recording of lectures and other events also weren’t yet a thing.

However, this wasn’t what a lot of potential students were looking for; they wanted a virtual version of a university course. As the 2000s progressed, many were becoming accustomed to watching instructional videos and lectures on YouTube. Various academics and others were also starting to see an opportunity to broaden the reach of elite university educations to populations that were underserved in their access to that level of education.

Precursors had been around for a while, but it was 2012 when MOOCs exploded including both venture capital-funded (Coursera and Udacity, both with Stanford professors attached) and non-profit (edX, initially begun by MIT and Harvard). As *The New York Times*’ Laura Pappano wrote at the time, “The shimmery hope is that free courses can bring the best education in the world to the most remote corners of the planet, help people in their careers, and expand intellectual and personal networks.”

Courses were free and sign-ups were often massive—up to 150,000 or so massive—although the drop-out rate was correspondingly high, with numbers in excess of 95 percent common. Not really surprising given the lack of any real commitment required to hit the register button.

MOOCs are still around but their record has been mixed.

From an open source perspective, they always seemed closer to free as in beer, rather than free as in speech. By design, most MOOCs follow a fairly traditional format for a university course with short talking-head and virtual whiteboard videos standing in for the lecture hall experience. Grades (remember extrinsic motivation) mostly come from various autograded tests and homework including multiple choice, numerical answers, and computer program output.

Courses often do encourage participation in discussion forums and they often enlist past students as teaching assistants. However, many courses don't follow a rigid schedule. Furthermore, the practical realities of holding meaningful discussions among thousands of students at vastly different levels of educational background and language ability make MOOCs much more of a broadcast medium than a participatory one, much less one where participants can choose what material is covered, learn from each other, or even guide the overall course direction.

For the VC-funded Udacity and Coursera in particular, the difficulties of making money through giving content away for free also became troublesome. A number of MOOCs started out by offering a “verified certificate”—using the same sort of tools used for online proctoring of other types of certifications—for a fee. The problem was that a lot of employers didn't see a lot of value in a verified certificate relative to an unverified one, assuming they saw value in completing a MOOC at all. Over time, MOOCs have generally come to make all grades and certificates paid add-ons; some have even eliminated tests and other exercises from the free version. In 2013, Udacity would “pivot” (in Silicon Valley jargon) to simply charging for classes and focusing on vocational training.

This last point reflects what much of the audience for MOOCs ended up becoming anyway. For example, in 2013, after six months of high-profile experimentation, San Jose State University “paused” its work with Udacity because students in the program actually were doing worse than those in normal classes.⁹ The student body of a typical MOOC tends to be populated heavily with early- to mid-career professionals, often with

⁹<https://www.insidehighered.com/news/2013/07/18/citing-disappointing-student-outcomes-san-jose-state-pauses-work-udacity>

advanced degrees. On the one hand, MOOCs continue to be a great resource for those educated and self-motivated learners. But they've been a bitter disappointment for those who saw MOOCs as a remedy for the barriers high prices and inefficiencies of traditional university education erected against those who lack money and family support.

Collaboration versus Consumption

There are those who argue that MOOCs were a bad model anyway.

In 2013, Rolin Moe wrote that: “Had Udacity been able to provide a modicum of quality education to underrepresented populations, criticism would have remained to argue the pedagogy of such learning. With Udacity shifting away from underrepresented populations, the criticism is now about what is left in the wake of 2+ years of hyperbole and 0 years of results. And we cannot confuse what has shifted. It is not the narrative but only their business model; the narrative is still a system in crisis and searching for some low-cost, tech-savvy gizmo to do the job because we only need 10 schools and unions are the problem and our kids don't know enough STEM and plumbers make more money than college grads anyway.”¹⁰

MOOCs essentially solved the problem of bringing a lecture to those who can't be physically present. But that's been a more or less solved problem since the advent of VHS tape. MOOCs make the lecture more consumable but it's still basically just a lecture.

Martin Weller has written about how connectivism as a learning theory “as proposed by George Siemens and Stephen Downes in 2004–2005, could lay claim to being the first internet-native learning theory. Siemens defined connectivism as ‘the integration of principles explored by chaos, network, and complexity and self-organization theories. Learning is a process that occurs within nebulous environments of shifting core elements—not entirely under the control of the individual.’” He went on to say that “What was significant about connectivism was that it represented an attempt to rethink how learning is best realized given the new realities of a digital, networked, open environment, as opposed to forcing technology into the service of existing practices.” Yet, “while connectivism provided the basis for MOOCs, the approach they eventually adopted was far removed from this and fairly conservative.”¹¹

¹⁰<https://allmoocs.wordpress.com/2013/11/15/udacity-shifting-models-means-never-having-to-say-youre-sorry/>

¹¹<http://blog.edtechie.net/pedagogy/25-years-of-edtech-2010-connectivism/>

My Red Hat colleague Gina Likins argues that we should be thinking about education in terms of broader participation. For example, “people should be expected to fork the educational materials. What one classroom needs can’t be expected to work in another. Students will come from different backgrounds with different experiences. The textbook is only the start.”

Likins also points to other examples of open source community development that could apply to education. For example, many educational materials—textbooks in particular—follow a philosophy of not releasing before it’s final. There are valid reasons for some of this. And, especially at the secondary school level, there are complex political considerations with some topics like history as well. But it’s another way in which educational resources are developed with feedback from only a relatively small insular group, often with a limited set of perspectives.

Academic research is also seeing open access movements. Some institutions have adopted open access policies to grant the public access to research materials. The Public Knowledge Project maintains an open source publishing platform called Open Journal Systems, which editorial teams can use to referee and publish (largely open access) academic journals outside the traditional publishing system.¹² It’s also become more common to publicly circulate drafts and other pre-print versions of research as a way of soliciting broader feedback.

Opening Hardware

Open source hardware has less of a cleanly established storyline than in the case of software. Indeed, specific industry open source hardware licensing initiatives (such as OpenSPARC and OpenPOWER) are arguably less—or at least less broadly— interesting than less well-defined “maker” activities in general.

Ham Radio

One early example of open source-ish sharing of hardware designs is amateur radio. The common term “ham radio,” as amateur radio came to be known, was actually born of a slur. Professional wired telegraph operators used it in the 19th century to mock operators

¹²<https://opensource.com/resources/what-open-education> CC-BY-SA

with poor Morse code sending skills (“ham-fisted”) and it carried over to the amateurs experimenting with wireless telegraphy at about the beginning of the 20th century.

Factory-built gear wasn’t readily available as ham radio was getting started, so amateurs began handcrafting vacuum tube-based transmitters and receivers. After World War II, surplus military gear also became widely available.

Amateur radio publications encouraged this sort of grassroots experimentation with hardware. In *Ham Radio’s Technical Culture (Inside Technology)* (MIT Press, 2008), Kristen Haring recounts how, in 1950, *CQ Amateur Radio Magazine* announced a “\$1000 Cash Prize ‘Home Brew’ Contest” and called independently-built equipment “the type of gear which has helped to make amateur radio our greatest reservoir of technical proficiency.”

This hobbyist homebrew culture gave rise to the once widespread Radio Shack chain. Started in 1921 by two brothers, Theodore and Milton Deutschmann, it provided equipment for the then-nascent field of ham radio from a downtown Boston retail and mail-order operation. The “radio shack” term came from a small, wooden structure that housed a ship’s radio equipment.

The company had ups and downs and ownership changes over the years before its owners largely shut it down in 2017. However, for a few decades after its acquisition by Tandy in 1963, RadioShack stores (as they were called by then) were popping up in shopping malls and city centers across the United States and elsewhere. And, in so doing, becoming a sort of home for electronics enthusiasts of all stripes, including early personal computer hobbyists. The TRS-80, introduced in 1977, was one of the first mass-produced PCs and initially outsold the Apple II by harnessing the power of RadioShack’s retail channel and its thousands of locations.

To be sure, some of the nostalgia is selective. The company may have been the most convenient place for what we now call “makers” to pick up a needed resistor or capacitor. But its ubiquity also made it the default for often less-knowledgeable consumers to pick up subpar audio products and other consumer electronics in the days when the alternative was usually either a specialty retailer or a department store.

A Shift from Making

RadioShack was doomed in large part by some of the broad changes in electronics retailing dating to roughly the beginning of the 21st century. E-commerce, big box stores, and the emergence of smartphones were all in that mix and RadioShack never really adapted.

However, there were also technology shifts happening that affected the DIY tinkerers who viewed RadioShack so fondly even while the company was still successful in the 1990s.

Through about the 1980s or so, it was possible to design, build, and share plans for nontrivial electronics projects more or less from scratch. The Heath Company sold electronic test equipment like oscilloscopes, home audio gear, TVs, amateur radio equipment, and more in kit form under the Heathkit brand. A certain satisfaction and knowledge (if often frustration!) came from soldering and hand-assembling a complete working device from components, even if you were just working from cookbook instructions.

Byte, a relatively mainstream computer magazine, carried a regular column by Steve Ciarcia called “Circuit Cellar” that introduced a variety of electronics projects that readers could build. As in the case of the code the computer magazines printed, there was usually no explicit license attached to these types of designs, but there was an implicit understanding that they were there to use, modify, and share.

The first thing to affect the traditional electronics DIYers was the personal computer. Many early PCs were something of a DIY project on their own. But it was more in the vein of assembling boards and other prebuilt parts such as disk drives and then figuring out why the thing wouldn’t work. The process involved problem solving and learning, but it was certainly different from creating from scratch.

But PCs, after you got one working, also drew many hobbyists from hardware to software. To many, software seemed to provide more opportunities to build “real” things. It was certainly easier to share your work with others who could benefit from it. Although open source software in a formal sense wasn’t very widely known in the 1980s, “freeware,” “shareware,” and source code published for educational purposes widely circulated on bulletin board systems and on disks sold at computer shows.

It was also the case that electronics were simply getting harder to tinker with. Components were miniaturizing. They were getting faster and more complex. The devices that most people were in a position to design and build at home looked increasingly primitive compared to what you could buy off-the-shelf, often for less money.

The New Makers

Things remained in more or less this state until the mid-2000s.

The Arduino project started in 2003 as a program for students at the Interaction Design Institute Ivrea in Ivrea, Italy. Its goal was to provide a low-cost and easy way for novices and professionals to create devices that interact with their environment using sensors and actuators. The project's products are distributed as open source hardware and software, allowing anyone to manufacture Arduino boards and distribute the software.

In addition, being open source, Arduino was significant because it offered a good model for hobbyists and other to practically build interesting hardware projects in the modern era. Arduino board designs use a variety of microprocessors and controllers and are equipped with digital and analog input/output (I/O) pins that may be interfaced to various expansion boards and other circuits (Figure 7-1). Effectively, an Arduino embeds and abstracts away a lot of the complexities involved with interfacing with the physical world.

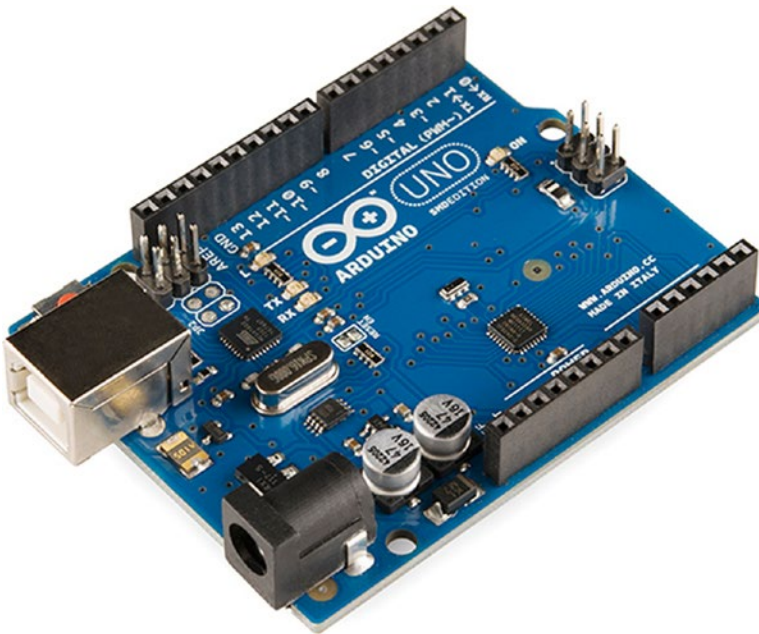


Figure 7-1. *Arduino microcontrollers can be used to build devices interact with the physical world through sensors and actuators. Source: Wikimedia. CC-BY-SA 2.0*

One person who recognized this was Limor Fried, who started Adafruit in her MIT dorm room during 2005 to sell electronics components and kits, including Arduinos. Now, it's a New York City company that she runs to make open source electronics kits and components for the growing tide of DIYers. For example, you can build a robot or a giant timer display.

In a 2011 interview with *Wired* magazine, Fried said that “One of the things about doing projects is that documenting them and sharing them with people used to be really difficult. Now we see so many people putting up videos, and it's so easy. I can make a five-minute video in an hour, with a preface and edits and nice audio and everything. And I think that makes a difference, because when people are exposed to this stuff and they actually get to see it, they get inspired.”

Another more recent hardware project is the Raspberry Pi, which is, in effect, a miniaturized low-cost Linux computer that can also interface with external components. First released in 2012, this series of small single-board computers was developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries. Unlike Arduino, the Raspberry Pi computer is not itself open source, but much of the software that it runs, including the operating system, is as are the plans for many projects that make use of the Raspberry Pi.

At about the same time as these computers for makers were becoming popular, buzz was also building for 3D printing. 3D printing is a subset of additive manufacturing in which material is selectively deposited based on the instructions in a 3D design file. Imagine thin layers of (typically) plastic printed one at a time until an entire three-dimensional object comes into existence.

3D printing is usually dated to Chuck Hull's 1986 invention of a stereolithography apparatus (SLA). However, from an open source and maker angle, the most relevant date is 2007, when Adrian Bowyer of the University of Bath founded the RepRap project, an open source project intended to build a 3D printer that could print most of its own components.

Over the next few years, commercial 3D printing started to roll out. One early company, MakerBot, pulled back somewhat from its open source roots—although more recently it's rolled out a Labs platform as a way to deal with criticism for its closed source stance.¹³

¹³<https://techcrunch.com/2017/10/18/makerbot-attempts-to-embrace-the-open-source-community-with-its-new-labs-platform/>

3D printing has proven popular as a way to economically create one-off parts. This is handy for a hobbyist to create a one-off case, gear, or just something decorative. However, in addition to its use in certain types of manufacturing, 3D printing is also used to create custom prosthetics and other medical devices.

Arguably, 3D printing hasn't yet lived up to the hype that's sometimes surrounded it, but it provides another example of how hardware designs can be cooperatively developed and shared.

Opening Culture in Organizations

More radically, we can even ask how and why organizations might change and evolve in a world where open source influences how individuals and organizations work together.

Why Do Organizations Exist Anyway?

"The Nature of the Firm" is a seminal 1937 article¹⁴ by Ronald Coase, a young British economist who would go on to win the Nobel Prize in economics over the course of a long career. Coase asked the questions: Why do people give up some of their freedom to work as they like and become full-time employees? And wouldn't it be more efficient for firms to depend on loose aggregates of private contractors, hired as-needed to perform a specific task? After all, orthodox economics going back to at least Adam Smith suggested that everyone should already be providing goods and services at the best rate available as part of an efficient market. Why hire someone who you might need to pay even at times when you don't have an immediate need for their skills and services?

His answer was that it's to avoid transaction costs associated with using a market-based price mechanism.

You need to find a person who can do something you want done but that you can't or don't have time to do on your own. You need to bargain with them. You need to trust that they will keep your trade secrets. You need to educate them about your specific needs. So, firms get created, and often grow.

Of course, it's a balancing act. Firms have always used specialist suppliers—think ad agencies for example—which can be thought of as reducing transaction costs themselves relative to contracting with individuals off the street. As *The Economist* noted on the

¹⁴"The Nature of the Firm" by R. H. Coase, *Economica*, Nov. 1937.

occasion of Coase's 100th birthday in 2010, however: "Mr Coase also pointed out that these little planned societies impose transaction costs of their own, which tend to rise as they grow bigger. The proper balance between hierarchies and markets is constantly recalibrated by the forces of competition: entrepreneurs may choose to lower transaction costs by forming firms but giant firms eventually become sluggish and uncompetitive."¹⁵

These ideas were built on by many economists over the years, including Oliver E. Williamson who once observed that it is much easier to say that organizations matter than it is to show why or how; he would also win a Nobel Prize for "his analysis of economic governance, especially the boundaries of the firm," which he shared with Elinor Ostrom.

More recently, New York University Professor of Law Yochai Benkler explicitly argued in a 2002 paper that "we are beginning to see the emergence of a new, third mode of production, in the digitally networked environment, a mode I call commons-based peer production." This third mode, the successor to firms in companies and individuals in markets, arose from open source software, he said.¹⁶

It's at least interesting to ask how online markets, connected communities, and the "gig economy" (think Uber) change historical equations. We've seen the rise of coopetition earlier in this book. There's little doubt that contracting out for certain types of work has indeed become easier for companies—for better or worse. On the other hand, while public clouds have become a generally beneficial option for certain types of computing workloads, there are also plenty of outsourcing horror stories in IT. We should perhaps leave this question at "it depends" and refer back to examples of how open source communities best work as the relevant discussion points for this book.

However, it's also worth considering what open source practices and principles mean *within* a given firm. Red Hat CEO Jim Whitehurst refers to this as the "open organization" in a book of the same name (*The Open Organization: Igniting Passion and Performance*, Harvard Business Review Press, 2015).

Open Organizations

General Motors might seem an odd starting point for this discussion, but, for all its various problems over the years, it provides an interesting study point about the principles of decentralization, which is a component of openness.

¹⁵<https://www.economist.com/business/2010/12/16/why-do-firms-exist>

¹⁶"Coase's Penguin, or, Linux and The Nature of the Firm," Yochai Benkler, 112 *Yale Law Journal* (Winter 2002–03).

Writing in 2018, Steve Blank noted how “Borrowing from organizational experiments pioneered at DuPont (run by his board chair), Sloan organized the company by division rather than function and transferred responsibility down from corporate into each of the operating divisions (Chevrolet, Pontiac, Oldsmobile, Buick and Cadillac). Each of these GM divisions focused on its own day-to-day operations with each division general manager responsible for the division’s profit and loss. Sloan kept the corporate staff small and focused on policymaking, corporate finance, and planning. Sloan had each of the divisions start systematic strategic planning. Today, we take for granted divisionalization as a form of corporate organization, but in 1920, other than DuPont, almost every large corporation was organized by function.”¹⁷

GM was still the epitome of a hierarchical organization, of course, in the vein of the companies that William Whyte wrote about in 1956 in his influential *The Organization Man* (Simon & Schuster). A central tenet of the book is that average Americans subscribed to a collectivist ethic rather than to the prevailing notion of rugged individualism. The *Fortune* magazine writer argued that people became convinced that organizations and groups could make better decisions than individuals, and thus serving an organization was a better logical choice than focusing on individual creativity.

GM was a hierarchical organization distributed through accounting structures rather than direct command and control.

Today, we see some signs of broader change.

More democratic forms of organizational governance exist. One popular discussion topic is holacracy, which introduces the idea of roles (rather than job descriptions), as part of a system of self-organizing, although not self-directed, circles. The term was coined by Arthur Koestler in his 1967 book *The Ghost in the Machine* (UK: Hutchinson; US: Macmillan). It’s now a registered trademark of HolacracyOne although the model itself is under a Creative Commons license. The best-known practitioner is probably Zappos, the online shoe retailer now owned by Amazon. In a 2015 memo, CEO Tony Hsieh wrote “Holacracy just happens to be our current system in place to help facilitate our move to self-organization, and is one of many tools we plan to experiment with and evolve with in the future. Our main objective is not just to do Holacracy well, but to make Zappos a fully self-organized, self-managed organization by combining a variety of different tools and processes.”

¹⁷<https://steveblank.com/2018/04/23/why-the-future-of-tesla-may-depend-on-knowing-what-happened-to-billy-durant/>

However, the prevailing wisdom—which reflects practices familiar to open source development practitioners—is not so much about democracy as it is about decentralization and empowerment based on skills and expertise.

In *The Open Organization*, Red Hat’s Whitehurst argues that “Many people assume that if an organization is not top-down that it must be some flavor of democracy—a place where everyone gets a vote. In both hierarchies and democracies, decision-making is clear and precise. It’s proscribed and can be easily codified. In most participative organizations, however, leaders and decision-making don’t necessarily follow such clear rules, just as it was in ancient Athens, where literally every citizen had the same opportunities to lead or follow others. Some people have more influence than others.”

One interesting aspect of such an environment is that it doesn’t necessarily mean, as one might assume it would, eliminating managers. Whitehurst writes that “Nothing could be further from the truth. Our managers play a vital role in building, supporting, and moderating the meritocracy. Finding that balance between supporting it and, at the same time, leaving things alone is critical.” Think back to our discussion of the role of maintainers like Greg Kroah-Hartman in the Linux kernel and this dynamic will seem familiar.

The need to delegate and federate decision making isn’t a new insight. It was pushed down to the divisional level at GM under Sloan. Management consultant Gary Hamel argues, “building and benefiting from communities at scale requires us to start with ‘openness’ as a principle, rather than with some particular set of collaborative tools or practices.” It’s more pervasive than an organizational delegating decision making within a formal structure.

In *Team of Teams: New Rules of Engagement for a Complex World* (Portfolio, 2015), Stanley McChrystal, who commanded the US Joint Special Operations Command in the mid-2000s, describes the basic problem. He recounts how the scientific management system that Frederick Taylor unveiled at the 1900 Paris Exposition Universelle “was so beautiful it inspired people to devote their lives to his vision.” It was so impressive for how efficient it was at executing known, repeatable processes at scale. His steel model could churn out metal chips at a rate of 50 feet per hour rather than the norm of nine.

However, McChrystal goes on to write, today’s world is more interdependent. It moves faster.

This creates a state of complexity, which is fundamentally different from challenges that are “merely” complicated in a technical sense. Complexity in this sense means less predictable. It means emergent behaviors that increasingly need to be reacted to rather than planned for in great detail.

McChrystal argues for pivoting away from seeing efficiency as the managerial holy grail to a focus on adaptability. He distinguishes commands rooted in reductionist perfection from teams that have a connectivity of trust and purpose that gives them an ability to solve problems that could never be foreseen by a single manager—even if they're less efficient in some theoretical sense.

All of which echoes the open source development model in so many ways.

Concluding Thoughts

Open source today is not peace, love, and Linux. OK. Maybe a little bit. Or even more than a little bit. The “free as in speech” aspect of open source is very much worth keeping in mind at a time when there's so much centralized control over social networks, search, personal information, and communication channels generally.

However, as we've seen throughout this book, you don't need to be a hippy to appreciate the value of open source.

It broke down the vertical silos of the computer industry and helped to prevent new horizontal ones from dominating all aspects of the computer industry. At a minimum, it's provided a counterweight to limit some potential excesses.

It's also proven to just be a very good software development model. While many successful projects have aspects of the cathedral to them (which is often both inevitable and necessary), the free-wheeling bazaar is also at least lurking. It removes friction associated with individuals and companies working together and has clearly influenced how even proprietary software development takes place in many cases.

Not all aspects of open source have been an unbridled success. Business models built around products that use only open source code have proven elusive for many companies. Some of today's largest tech companies make extensive use of open source for their online services but give little back into the virtuous cycle feeding open source development with the dollars from business value. There are also just vast swaths of the software universe, such as industry-specific applications, which haven't seen much historic open source software development.

However, as we've seen in both this chapter and throughout the book, open source reflects and informs changes in how individuals come together in order to innovate and otherwise accomplish missions that are important to them. Like most change, it's evolutionary. But organizations are cooperating more. They depend on each other more. We see more standards and therefore better interoperability. Information is more widely

shared and collectively created whether educational resources or knowledge more broadly. Even data to create physical artifacts can be exchanged.

And it's encouraging to believe that the values and thinking that gave birth to open source are being absorbed into business and culture more broadly. That may be hard to believe if you carefully follow the news headlines. But there are, in fact, significant qualitative differences between how successful organizations operated in decades past and how many do today. Transparency is greater. More decision making is decentralized.

The patterns are uneven. The future is still unevenly distributed. However, open source has taken a large bite out of software and of culture even as software is eating the world.

Index

A

Advanced Message Queueing
Protocol (AMQP), 62
Advanced Micro Devices (AMD), 10
Amazon Web Services (AWS), 122
Application programming
interface (API), 86
Allen, Robert C., 82

B

Benevolent Dictator for Life (BDFL), 54
Berkeley System Distribution (BSD), 4
Bloch, Peter, 71
Bulletin board system (BBS), 6
Business models, 105
 advantage of, 109
 aligning incentives, 114
 balance right, 107
 categories of, 107
 coopetition—cooperative
 competition, 115
 beneficiary and catalyst, 117
 coopetition, 115
 interoperable solutions and
 standards, 116
 standards, 118
 core competencies, 113
 Cygnus solution, 111

DevOps, 120
 abstractions to separate
 concerns, 122
 Agile principles, 122
 manufacturing analogs, 124
 overview, 120
 site reliability engineers, 123
distributions/distros, 112
freemium model, 108
GNU system, 106
independent software vendor, 111
open core *vs.* open source, 108
open source and DevOps
 approach, 128
 culture, 129–130
 incentives, 128
 iteration, experimentation and
 failure, 126
 platforms and tooling, 124
 process, 125
 scope, 128
 shifting culture, 129
 workflow, 128
pervasive open source, 130
speed
 consumerization, 119
 physical to virtual, 119
subscriptions, 113, 114
subscriptions and support, 110

C

- Central *vs.* distributed control, 50
- Cloud Native Computing Foundation (CNCF), 56
- Coase, Ronald, 168
- Collaboration and communication, 85
 - contributors interact, 89
 - limits of communication, 86
 - modularity, 88
 - software structure, 87
- Collective invention, 82
 - advantages of, 84
 - Allen, Robert C., 82
 - economics, 83
 - protective measures, 85
 - sharing of knowledge, 85
- Computer Systems Research Group (CSRG), 4
- Consensus, 55
- Continuous integration/continuous deployment (CI/CD) pipeline, 125
- Contributor license agreement (CLA), 109
- Core Infrastructure Initiative (CII), 39

D

- Defense Advanced Research Projects Agency (DARPA), 118
- Development model, 49
 - approaches, 50
 - BDFL, 54
 - caveat, 51
 - central *vs.* distributed control, 50
 - coders, 61
 - direct involvement of, 62
 - users become contributors, 62

- community
 - committers, 60
 - contributors, 60
 - leaders, 59
 - maintainers, 59
- consensus, 55
- contribution
 - anti-patterns, 63
 - culture, 66
 - formal mentoring programs, 65
 - friction of tools, 64
- decision making, 52
- determining, 72
- effective communications, 68
 - bazaar, 76
 - community, 78
 - courses, 75
 - distributed teams, 69
 - duplicative, 77
 - free-for-all, 77
 - limits of, 69
 - measurements, 72
 - metrics, 73
 - number meaning, 74
 - nurturing conversations, 72
 - quantity and quality, 73
 - sabermetrics, 73
 - softer aspects, 75
 - team and community dynamics, 70
 - virtual, 70
- explicit and community governance model, 52
- foundational principles
 - business and technical governance, 57
 - cloud-native workloads, 56
 - community members, 58
 - goals, 57

- hardware, clouds and containers, 58
- innovative platform, 58
- open first, 56
- software developers, 58
- maintaining community, 66
 - documentation, 67
 - modular beats monoliths, 68
 - responses, 67
- meritocracy, 55
- overview, 49
- Drives open source, 81

E

- Easy On-Ramp, 67
- Ecosystem technologies (ET), 147
- Electronic Data Interchange (EDI), 21
- Encouraging and rewarding
 - innovation, 83
- Enterprise resource planning (ERP), 127

F

- Flip side, 131
 - development model, 145
 - ecosystems, 146
 - four-layered IT model, 147
 - open source code, 149
 - vertical integration, 146
 - gravity shift centers, 144
 - hybrid, 143
 - positive feedback loop, 131
 - business value, 133
 - cloud computing, 134
 - cloud matters, 136
 - contributing code, 139
 - Hoover Dam, 136
 - IT industry, 133

- money, 138
- out at the edge, 137
- solutions and products, 132
- upstream projects, 132
- users concept
 - bundles, 139
 - convenience, 141
 - hiding complexity, 142
- Free and open source software, 1, 25
 - accelerates, 16
 - build/buy, 18
 - disruptive innovation model, 19-20
 - ecosystems, 21
 - enterprise IT model, 17
 - Linux and open source, 22
 - monoliths, 21
 - status Quo, 18
 - web, 17
 - Ah, Unix, 2
 - AT&T's specific case, 3
 - breaking community, 6
 - coining of, 26
 - commercialism, 27
 - commercial version, 3
 - commons, 29
 - copyleft mire, 30
 - distribution, 30
 - driving participation, 31
 - foundations, 8
 - free software enters, 7
 - GNU Manifesto, 26
 - hardware and software, 9
 - mass-market operating system, 11
 - Microsoft Swings, 11
 - NetWare, 10
 - vertical silos, 9
 - Windows NT, 12
 - IBM 704 computer, 1, 2

INDEX

Internet
 operating system, 14
 scale-up to scale-out, 13
licenses, 28
Linux, 14
 eclipse, 16
 grows, 15
 *nix, 15
open source compliance
 controls, 32
 ongoing process, 33
 policies, 32
OSI serves, 28
PCs, 5
permissive licenses gain, 30
pragmatism, 27
projects
 approach, 40
 commercial software products, 42
 demarcation, 41
 doubling down an existing
 project, 43
 OSPO, 46
 Zemlin, Jim, 41
projects *vs.* products, 33
 intersection, 36
 reducing risk, 36
 support means, 35
 upstream and downstream, 33
securing open source
 hurts-security side, 38
 patches and advice, 37
 risk, 39
 secret sauce, 38
 vulnerabilities, 38
sharing code, 2
Stallman, Richard, 25

G

Garzarelli, Giampaolo, 88
General Public License (GPL), 28
Gilmore, John, 111
GNU Public License (GPL), 8

H

Hartmann, Richard, 134, 135

I, J

Independent Software Vendor (ISV), 111
Innovation, *see Collective invention*
Internet Protocol (IP), 118
IRC and Slack messaging tools, 70

K

Kubernetes (CNCF), 42

L

Langlois, Richard N., 88
Linux, 112
Linux Foundation, 42

M, N

Massive open online courses (MOOCs), 160
McChrystal, Stanley, 171, 172
Measurement
 behaviors, 96
 broader view, 103
 CHAOSS metrics, 102
 community health, 100
 cross-purposes, 97

- cultural emphasis, [101](#)
- DevOps model, [98](#)
- direct measures, [97](#)
- diversity and inclusion, [101](#)
- meaning, [96](#)
- metrics, [98](#)
- motivational measurements, [97](#)
- project growth-maturity-decline, [102](#)
- risk, [102](#)
- stability measurements, [99](#)
- throughput measures, [99](#)
- value, [102](#)
- Meritocracy, [55](#)
- Microsoft Swings, [11](#)
- Miller, Matthew, [58](#)
- MIT OpenCourseWare, [159](#)
- Motivation
 - extrinsic, [93](#), [95](#)
 - internalized extrinsic motivators, [94](#)
 - intrinsic, [93](#)
 - researchers, [93](#)
 - types of, [92](#)

O

- O'Grady, Stephen, [141](#)
- Open educational resources (OER), [160](#)
- Open Source Initiative (OSI), [28](#)
- Open source program office (OSPO), [45](#)
- OpenStack projects (OpenStack Foundation), [42](#)
- OpenStreetMap, [153](#)
- Opportunities and challenges, [151](#)
 - data and computer programs, [152](#)
 - education focus
 - collaboration *vs.* consumption, [162](#)
 - MIT OpenCourseWare, [159](#)

- MOOCs, [160](#)
- precursors, [159](#)
- hardware
 - Arduino microcontrollers, [166](#)
 - 3D printing, [168](#)
 - ham radio, [163](#)
 - makers, [166](#)
 - technology shifts, [165](#)
- independent contribution, [158](#)
- organizations
 - democratic forms, [170](#)
 - General Motors, [169](#)
 - gig economy, [169](#)
 - hierarchical, [170](#)
 - McChrystal, Stanley, [171](#), [172](#)
 - overview, [168](#)
- ownership of data, [155](#)
- principles and practices, [151](#)
- read/write web, [156](#)
- transparency through data, [153](#)
- Wikipedia, [157](#)

P, Q

- Participation, [90](#)
 - mentoring, [91](#)
 - onboarding process, [91](#)

R

- Reflecting and informing, [104](#)

S

- Service-oriented architecture (SOA), [22](#)
- Site reliability engineer (SRE), [123](#)

INDEX

Special Interest Groups (SIGs), [43](#)

Stability measurements, [99](#)

Stallman, Richard, [7](#), [25](#)

T

Tensor Processing Units (TPU), [152](#)

Tiemann, Michael, [111](#), [113](#)

Transaction Workflow Innovation
Standards Team (TWIST), [62](#)

Transmission Control
Protocol (TCP), [118](#)

U

Unix, [2](#)

US Geological Survey (USGS), [153](#)

V

Viteles, Morris S., [100](#)

W, X, Y, Z

Wikipedia, [157](#)

Windows NT, [12](#)