Building a Business on Open Source

A manager's guide to efficient, effective, and profitable software products and services

By John Mark Walker

THE LINUX FOUNDATION

Table of Contents

About the Author

John Mark Walker is a noted open source product, community, and ecosystem expert and has built numerous open source communities, launched new product initiatives, and implemented collaborative processes that yielded more efficient product development and higher innovation. You will often find him in the halls of various open source conferences, as he much prefers the hallway track to actual talks, talking to people doing interesting things. A recognized thought leader, he wrote the seminal article "There Is No Open Source Community" and has spoken at numerous conferences on the subject of open source community engagement and product strategy. John Mark recently started the web site OSEN (osenetwork.com), where readers can learn more about creating products and services with open source software: supply chain efficiency, open source product management, innersource principles, and the value of increasing your employees' open source participation.

When he's not thinking about this stuff, John Mark lives on the outskirts of Bostonia with his wife, Cathy, and their two children, Jackie and Cary.

Introduction

I never thought I would need to write this book, certainly not in 2017. See, way back in 1999, when this whole free software and open source thing started to really take off, I figured I knew what would happen: there would be a ton of open source software, it would become the way people worked, and there wouldn't be a need for experts to explain to others how to do it, because everyone would know how. OK, so two out of three isn't bad. Fast forward to the present: Open source *is* everywhere, it *is* how innovation happens, and yet… it seems that there's still a need to explain how to do it well.

This is partially because doing things the right way is never easy and requires a never ending quest for improvement. And it's partially because our standards for good open source practices have evolved over time. It used to be that the very act of taking source code and posting it on a website was considered an act of rebellion. A "crazy" thing for a company to do. "Give away your source code? Are you crazy?" Or, if you wanted to assert your hippie, activist bona fides, working on free software was the perfect way to establish your credibility. "Hey man, I'm here to help others with software, and like, I totally don't need money, man!" Many of those things turned out to be wildly inaccurate, but mostly, we have developed a more refined understanding of how people collaborate in teams.

Back in the day, "product" was a dirty word. There were many who thought that by developing software for free, they were undermining or subverting the very act of creating product. They felt this was an act of anarchy, destroying the viability of large technology behemoths by giving it away for free. Some tech behemoths were actually destroyed or decimated in the process, because they didn't take it seriously. But the idea that product was dead and buried turned out to be false. There was significant change in how products were developed and sold, but some things didn't change: customers will still pay for value, even if they can get the source code for free. Any product can be built with source code, but good products are about way more than source code. At a minimum they're about testing procedures, integration, packaging, risk management, manageability, support, and delivering solutions that work as promised.

Open source products have gone through many different iterations over the years. Back when everyone thought source code == product, there was a terrible tendency to think of open source projects as being cheap knockoffs of premium products. The proprietary products were where the innovation happened, and open source projects would copy functionality on the cheap. The first form of products were CDROMs you could use to install software, and then you could buy support. Think Red Hat Linux, before Red Hat turned the corner with Red Hat Enterprise Linux. And then there was a series of startups based on a simple premise: give away some limited version of a premium product for free, under an open source license, and then sell the premium product. Later, with more open source projects finding success, there was a movement to sell hybrid products with a core open source platform and proprietary management software on top. Now, with so much software developed by companies that don't sell software, we see companies across a number of industry sectors selling services built with open source software. There is more than one way to do it, but some ways work better than others (more on that in subsequent chapters).

The open source ubiquity of today has led to a different kind of problem. While everyone is seemingly developing some type of open source code, the ease of building software with other people's software has led to a general sloppiness and lack of product discipline. I know I sound like the old guy on the porch when I say it, but software development has become too easy, and not enough developers or IT pros think about where the software comes from and how to ensure it works over the long term. If you're building a software product with open source components, how can you vouch for its worthiness? How do you know if it will continue to be developed? What happens if the developer abandons the project? How do you calculate the risks inherent in upstream open source projects and account for them when delivering your products to customers?

This book is for anyone who needs to create a reliable software product or service for their customers. The intent was to write about products, services and/or solutions that are sold to customers for a price, but the principles found in these chapters apply equally well if your customer is internal. What I found is this: if you need to develop and sell a product, you need to think about the value of the thing you're selling. Hint: source code is not valuable. A working solution is valuable. In that sense, it doesn't matter whether your code is open source or not, because all that matters is whether the thing you're selling actually provides value to the buyer. However, if you're developing software in an open source way, you have options that proprietary developers don't have. You can deliver better software more efficiently that is more responsive to customer needs — *if* you do it well and apply best practices. This book is about how to do that.

Part 1

What is an Open Source Product?

Chapter 1
Intro to Open Source Business Models

By now, I thought it would be self-evident how to derive revenue from open source software platforms. But alas, no. Despite the fact that the success of open source software is unparalleled and dominates the global software industry, there are still far too many startups repeating the same mistakes from a thousand startups past. And there are still far too many larger companies that simply don't understand what it means to participate in, much less lead, an open source community.

As of early 2017, it's been about 19 years since "open source" was coined to replace the term of art, "free software" — deemed far too scary for the MBA-toting, pointy-haired bosses among us:

"Freedom? Yech! Let's use open source instead."

Since then, every single area of innovation in computing is now dominated by the open source platforms among us. From operating systems, virtualization and containers to big data, cloud management, and mobile computing, chances are good that innovation is happening on an open source platform. So dominant is the breadth of open source ecosystems that virtually the entire world's economy depends on open source software. I hesitate to think what would happen to the global economy should open source software be suddenly made inaccessible (think financial collapse and societal unrest on a biblical scale).

Despite this massive success, there is something missing from the world of open source: successful open source companies. We are now in a "Tale of Two Cities" type of story: It was the best of times, it was the worst of times. Many of the most visibly "successful" open source startups from the last ten years are companies that were acquired by larger ones. These companies, while successfully attracting large communities of users and developers, were not sustainable, revenue-wise. There are a number of executives from

open source-heavy software vendors who have openly remarked that it is impossible to create a successful business with a “pure” open source approach.

WHAT VC INVESTORS WANT

Many companies have adopted a particular model around creating and selling open source software: create an open source platform or base and sell proprietary software that integrates with the underlying open source platform. In this case, the platform is always free or “Free” and the product they actually sell for revenue is proprietary. This model has been labeled “Open Core” — more on that, below — and is the way most VC-backed companies approach open source development.

In fact, there has only been one company (yet) to ever make a viable business model from selling only open source software: Red Hat. But is that the end of the story? Is it really as simple as “nobody can create another successful pure open source business?” I submit that those who agree with that statement are wrong. For one thing, the Open Core approach has never resulted in a successful product. The reason companies do this is that, in their opinion, this allows them to increase their profit margins and more quickly approach profitability. However, most of the value in this model is in the platform itself, and treating it as a worthless commodity while simultaneously trying to build proprietary software on top results in a cut-rate buyer’s market without the margins the vendors hoped for. Furthermore, this approach leads to a constant battle between “how much do we cripple the open source platform to enable more proprietary sales” and “how can we make the open source platform good enough to increase adoption, but not too good.”

No proprietary product built as an addition on top of open source software has ever achieved ubiquity in the modern data center, with the possible exception of VMware, which utilized a stripped-down version of Linux for its ESX hypervisor. Even then, the Linux base for ESX was a pretty small part of the overall platform VMware built. It seems that the only successful products that utilize open source components are those where the proprietary bits are the platform, and the open source parts are simply the commodity bits that fill in the gaps that developers can use to more quickly create a product.

SHIFTING BUSINESS MODELS

And yet… purely open source systems are proliferating in every industry in every region of the world. The list of successful open source platforms grows every day. How does

one follow the money? How is this even possible if vendors are right when they claim that they cannot build successful businesses with a pure open source approach? The problem is not that software vendors cannot make money selling open source products, but rather that the problem is misstated. For one, we are looking in the wrong places: software isn't created only by software vendors anymore. There are consultants, systems integrators, in-house developers, cloud hosting providers and a host of other people who create, buy, sell and use open source products.

It is not that there is no money in selling open source software, but rather that the business models have shifted. Whereas, under the old proprietary world, a larger percentage of money went to pure software vendors, now that money has spread among a larger spectrum of companies and industries; lots of people get paid to work on or with open source software, but an increasing number of them don't work for software vendors, per se. In addition to looking in all the wrong places, the current investment model is suspicious of an open source approach. The vast majority of venture capitalists, especially in Silicon Valley, are very risk averse and shy away from open source products that, in their view, will not give as large a return on their investment. In order to secure the funding required to scale a company, investors will frequently require that the startup company include proprietary bits as tools to increase revenue and margins. These two factors — diffusion of revenue and risk-averse investors — combine to both give a false impression and, in part due to the false impression, prevent pure open source software vendors from getting funding.

It is not that there is no money in selling open source software, but rather that the business models have shifted.

SERVICES AND SUPPORT

Quite a number of companies offer services and support around open source technologies, either as an adjunct offering to complement their products, or as their primary source of revenue. However, there are a few reasons why I chose not to emphasize this particular model in this book:

- **It doesn't scale**
  I wanted to restrict the subject matter to ambitious business models that could, eventually, grow into a global scale. Services and support don't necessarily prevent that, but it becomes much more difficult. If you have money to burn, scaling out such an operation is feasible. If you don't, it's pretty much impossible.

- **Investors don't like it**
  See bullet point above. As much as I have a love-hate relationship with the VC community, I can see their point in not investing in services and support companies. Investors want companies that will give them a significant return on their investment, and building a services company is a long slog — longer than with software products.

Some think that services and support are the only way to build a business on open source software, and this book is an active attempt to counter that.

- **Mom and Pop**
  If you want to create a viable business with a couple of people and not much more, and you have some runway before you have to turn a profit, a services model is great, especially if your team has sought-after skills that aren't very commonplace. For anything more than that, you need to seriously think about how you can grow this model or whether it's a good idea. Unless of course, you're simply using your services cash flow to fund product development, which is a strategy that can work for startups before they've signed their first term sheet.

So yes, there are quite a number of companies making money using business models not explored here in depth. But for a variety of reasons, they aren't particularly interesting to me. The focus here is on product. Specifically, that one can make a profit on open source products, which is something that seems to be in doubt. In fact, there are some who think that services and support are the only way to build a business on open source software, and this book is an active attempt to counter that line of argument.

Is there a future in pure open source products? And if so, how can you capitalize on it? In the following chapters, I will investigate many open source business models, some of which have been more successful than others. I'll also go into some level of detail on how to productize software derived from open source projects. It's both trickier and simpler than what you might imagine. And finally, I'll discuss the role of the software supply chain in open source and how companies can manage and influence it to best benefit their products and businesses.

Chapter 2
Open Core vs. Hybrid Business Models

Before I go any further, I need to make a disclaimer:

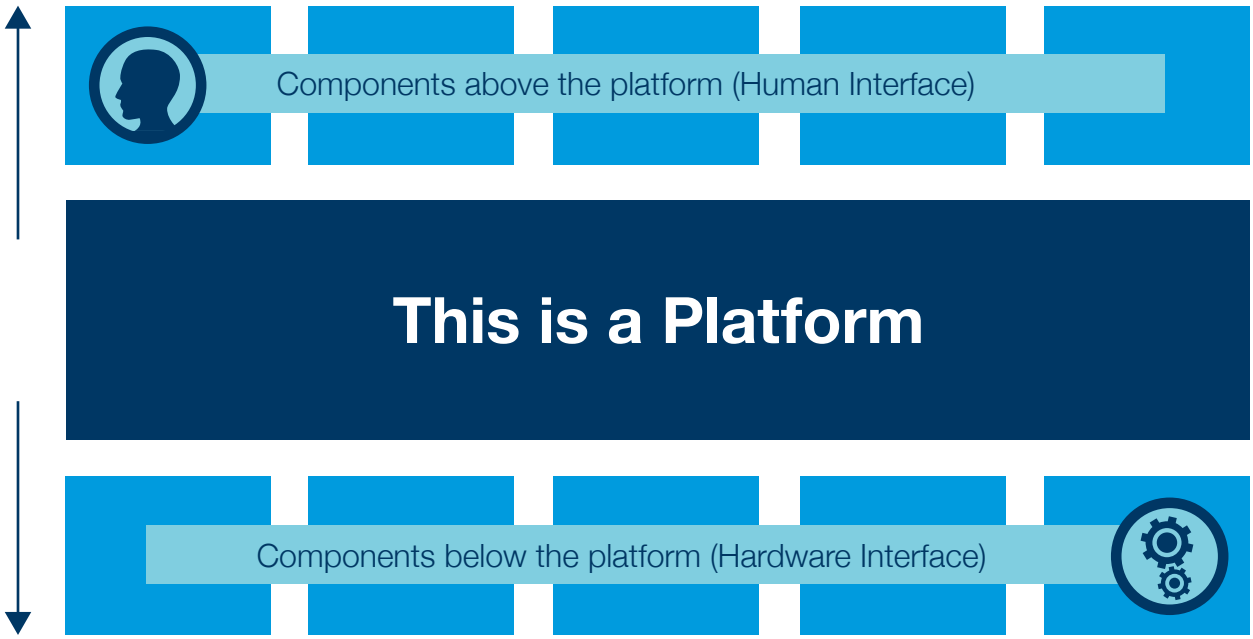
The opinions expressed in this book are mine alone and not those of my current or former employers. The insights in this series are gleaned from many experiences I've had over the last 16 years working with a wide variety of products, projects, communities, companies, developers and end users.

In the previous chapter, I dove into a discussion about open source platforms without actually defining what they are. I will remedy that now.

WHAT IS A PLATFORM?

Platforms are tricky things. Lots of companies aspire to create a world-beating software platform, and it's easy to see why - a platform implies something that supports lots of other things that stand on top of it. Take away the platform, and the whole house of cards falls flat. At least, that's what lots of software vendors want you to think, hence the rush to describe their products as "platforms."

In software, a platform isn't necessarily something that holds other things up. To oversimplify a bit, it's something that connects multiple pieces together with some type of logic and rules. These pieces are usually described as being "above" or "below" the platform, with "above" denoting software components that are closer to a human interface and "below" being software components further away from human interaction (and closer to the hardware interfaces). Other descriptions include "northbound" vs. "southbound" or "up the stack" vs. "down the stack." See the next page for a hopefully helpful diagram:



Caption: Figure 1: A software platform connects multiple pieces together with some type of logic and rules. These pieces are usually described as being "above" or "below" the platform, with "above" denoting software components that are closer to a human interface and "below" being software components further away from human interaction (and closer to the hardware interfaces).

Lots of things could fill the platform box, above, including the Linux kernel and Linux-based distributions, to name just two. I have a theory: the more components that need to be managed by a platform without human intervention, the more likely that platform is to be open source. The corollary would be: the more a platform requires human interaction, the less likely it is to be open source. I believe this is why WYSIWYG tools are more likely to be proprietary, whereas core infrastructure software, such as many of the tools that make up cloud computing services, are open source. This will help inform much of the discussion that follows.

Now that we understand a bit more about platforms, let's continue with some analysis of various open source approaches taken by software vendors.

THE OPEN CORE APPROACH

As mentioned in the previous chapter, "Open Core" is basically a proprietary approach with some open source software thrown in for sales and marketing lead generation. Now I will add some more detailed analysis about this approach.

Many venture-funded startups with enterprise software product ambitions are directed away from the pure open source approach almost from the beginning, as soon as they start to take seed capital and certainly by the time they take an “A” or “B” round. The idea is simple: if you have software that can be loosely construed as a “platform,” you should, in order to maximize your revenue stream, give away the platform and make money on proprietary additions to the platform. With ubiquitous platform adoption comes the ability to capitalize on the sales of add-ons, apps, plugins, extensions and even vertical market products based on the open source platform. It sounds simple enough and is certainly less scary than the prospect of selling pure open source code that can be obtained for free. There are several advantages and dis-advantages to this approach. First, the positives.

Many venture-funded startups with enterprise software product ambitions are directed away from the pure open source approach almost from the beginning.

OPEN CORE: ADVANTAGES

The most important aspect to an open core approach, for better or worse, is control of the platform ecosystem. In this model, the open source platform becomes a freemium loss leader for the other products sold around it. This entices the company that created it to keep it tightly constrained. With constraints comes almost total control over the open source community and ecosystem that spring up around it. This means that almost all core engineers will work for the vendor responsible for the platform. Consequently, the application ecosystem around the platform will consist almost entirely of the efforts of the platform vendor. This means that this vendor can control a near monopoly of the revenue derived from this ecosystem.

The goal in this scenario is pretty straightforward: create a large enough user and developer base around the platform, and a vendor can lay claim to a large potential customer base, at which point it’s the vendor’s job to convert some percentage of them into paying customers. The bottom line is that an open core approach is a siren song to investors and entrepreneurs, luring them with its promises of revenue and world domination.

OPEN CORE: DISADVANTAGES

There is a flipside to this quest for total control, and it comes at the expense of the advantages normally conferred from an open source development model. By controlling all the developers and consequently almost the entire ecosystem, there is a strong tendency towards creating a crippled open source platform. The goal becomes creating a ubiquitous platform that's good enough to spread far and wide to millions of users and developers, but not so good that it encourages them not to buy the "real" product. As a result, there's a constant tug-of-war between opposing factions within the vendor as to what should be part of the proprietary applications and what crumbs to leave for the "freeloaders." Various factions win out depending on how good the revenue is looking for a given quarter or fiscal year. There's also a tendency, in this quest for control, to want to own the copyright of all code so that it can be licensed commercially and sold as part of a product. Thus, anyone who wants to contribute to the project must ordinarily assign copyright to the vendor, further limiting the potential contributor base.

In the end, an open core platform is functionally the same as a proprietary freemium upsell model, with very modest benefits from the open source project: very few contributions, almost none to the core code base, and only participation on the fringes. In fact, the approach is so limiting on the open source side of the house, that one wonders why the vendor bothered to open source any code at all. With a diminished upside for the open source code comes a diminished number of potential customers, resulting in diminishing potential revenue. In fact, the open core model seems like a way to guarantee that the resulting product will be limited in scope.

Open core was all the rage in the mid-2000s, when several software vendors were able to score millions of dollars in venture funding with that strategy. What happened next was not pretty: many failed spectacularly, some limped along for years, and a very lucky few were acquired. There has, to date, never been a successful open core company. That is, none have ever reached profitability with their own products.

There has, to date, never been a successful open core company. That is, none have ever reached profitability with their own products.

Above-mentioned limitations also served to limit the adoption of the open source pieces, severely restricting their influence to only a very few number of projects and products. Even in those projects that did have a significant influence, ie. MySQL, they were bailed out by spectacular acquisitions. While they were likely never destined to be profitable, they were valuable to their suitors because of their large user bases. In those cases, the respective startups were much better about pushing a fully-functional free version but still found capitalizing on products derived from them to be a difficult task.

The primary problem with open core is that, in addition to being very difficult to capitalize on, it left a bad taste in the mouth of investors, leading them to conclude that this open source thing isn't very profitable. Indeed, the number of funded pure open source software vendors has declined in recent years, and the takeaway from the investor and even the entrepreneur community is that "open source doesn't pay." The result is that whenever a company shows up with an open source product, they are immediately pushed into the open core model, even with its history of failure, or a hybrid approach, which I'll describe in detail in the next section. The fundamental question to the open core approach is thus: Even with a highly successful open source "core," is there a means to drive enough revenue from the derivative product to sustain the growth of a company? The answer, thus far, is "no."

A HYBRID APPROACH

In the wake of the open core failures, the next generation of open source software vendors took a slightly different approach that attempts to combine the best of the open source and proprietary worlds. Instead of creating a severely limited (and probably unsuccessful) open core platform, the newer open source software vendors have been much smarter: create proprietary products or services based on collaboratively-developed open source platforms. This approach carries several advantages over open core. Instead of working so hard to spread a limited platform not usable by most people, take a successful open source platform and build applications, services or management on top of that. These applications can be proprietary, open source or even open core, but the platform from which they derive their relevance is fully open source and open to all collaborators. This approach is popular with many companies that have built tooling around various pieces of the Apache Hadoop and OpenStack ecosystems. At the center of this approach is an open source platform that is produced in a truly collaborative sense, open to all comers and, most importantly, nurtures the seeds of innovation that come from a variety of sources.

There are quite a few companies currently attempting to create products around these and other ecosystems, and time will tell which of them will be successful. There are open

questions as to whether a proprietary product, even one built on top of an open source ecosystem of projects, can, much like its open core cousin, be successful enough to sustain a growing company. The early results suggest that it's quite challenging, although it's too early yet to say one way or the other. There are certainly benefits from such an approach, but subject to some of the same limitations as open core. The final results are not in, but we have yet to see a company fully succeed on its own, without acquisition, and turning a profit that sustains growth over time. I suspect that the difficulties of the open core approach will rear their ugly heads here, too. If the open source platform is given away for free, are all derived products seen as limited offerings that don't offer enough value to command high revenue from customers? I don't know yet, but I have doubts about the approach.

The beauty of the hybrid approach is that the base open source platform is not limited in scope and can build a strong community identity. The downside is that it is not yet clear if simply adding management or other pluggable pieces on top of an open source platform is enough to create a product for which a vendor can charge significant revenue. I'm drawing a distinction, somewhat arbitrarily, between the above and engineering teams that consume open source libraries as they build out proprietary products. The former derives its core functionality from open source platforms and the latter merely plugs in open source components where appropriate. That's a well-established model and has many successful (and unsuccessful) examples to refer to.

OPEN SOURCE PLATFORMS AS A PRODUCT

My biggest problem with the open core and hybrid models described above is that they both assume there is no intrinsic value in the platform itself. (Pay attention! This is probably the key to the entire book.) In both the open core and hybrid models described above, it is assumed that no one will buy the platform and that all product value stems from the additional applications, whether proprietary or open source, that the vendors have created and applied to the platform. In other words, these two models assume that open source software is composed only of

> The open source platforms themselves are inherently valuable and can be sold as products in their own right, if done correctly.

commodity bits that no one will pay for; that anything open source is not something worth spending money on. I believe that assumption is false.

Will customers pay for an open source platform without proprietary applications or tooling? Everyone who invested in or started an open core or hybrid application company seems to think not, but let's consider the evidence. If open core and hybrid approaches have never actually delivered an outright success story, does it not lead one to believe that perhaps both approaches are lacking? If that is true, and if open source is truly at the center of data center technology innovation, which I believe it is, what does that leave? I am not discounting the added value of proprietary software on top of open source platforms; I am suggesting that the open source platforms themselves are inherently valuable and can be sold as products in their own right, if done correctly. More details on that in the next chapter!

Chapter 3
Creating a Product

What is the value of an open source platform? Would someone ever pay for it outright? Indeed, how does someone use an open source platform? Let's start with the oldest and most significant of open source platforms, Linux. For the longest time, Linux was dismissed as a non-viable data center technology for "enterprise-grade" or "business critical" operations because it had no support model, no applications that ran on it and no obvious way to make money from it. How, then, did Linux become the engine that fueled the growth of the world's open source ecosystem, an ecosystem that could be valued in the trillions of dollars, when calculating the percentage of the world's economy that relies on open source systems? Was it just a bunch of hippies sharing the software and singing about it, or were there clear business reasons paving the way to its eventual victory?

If we flash back to 1999 or 2000, it's sometimes difficult to remember that Linux was, while ascendant, very limited and not the runaway juggernaut we know of today. There were many Linux distributions that packaged Linux for end users, but none of them could boast of a product that made any substantial money. They were mostly packaged in boxed sets and distributed through big box stores, unless you were lucky enough to have a big enough internet connection to download your own copy. The idea of subscription-based enterprise software had not yet landed, and the only business model that most people understood for open source software was product support and services, neither of which were particularly easy to scale, especially for startup companies with limited resources.

It is in this context that we examine how some enterprise-class open source products started to make headway and achieve something resembling success. How did this happen? One thing is clear, the successful open source products never ever discount the intrinsic value of a scalable, world-class, reliable open source platform. Open source software may be usable, or it may not, but it's open source bona fides have nothing to

do with its usability. Or consistency. Or reliability. Or manageability. And most enterprises have a different view of "usable" from your average computer enthusiast or hobbyist. For an enterprise, "usable" means being able to achieve scale without having to resort to a whole lot of customization or private consulting. It should "just work" and fit neatly within an enterprise's existing workflow. Simply releasing the source code of an open source project does not imply any kind of guarantee of usability. Sure, enterprises can throw resources into making an open source project work for them, or they can pay someone who's already created a product that can do that more quickly.

OPEN SOURCE: IT'S ABOUT WAY MORE THAN CODE

This brings us to the most important question regarding open source products: how do you differentiate between the code that's available to everyone and a product derived from that code? This is the part that everyone gets hung up on and often leads to poor decisions. After all, if everyone has the code, then they have access to everything they need to run it, right? Thus obviating the need for a product? Not necessarily. Creating a product is a messy, messy business. There are multiple layers of QA, QE, and UX/UI design that, after years of effort, may result in something that somewhat resembles a usable product.

What the hybrid approach mentioned in the previous chapter gets right about open source is that you can't inject artificial limitations on an open source project and expect it to grow into an ecosystem. What it gets wrong, however, is the assumption that no one will pay for the base platform. A vendor with this approach assumes that anyone can install the platform on their own and will never need to pay for it. This is simply false. Getting a platform that is certified against an array of complementary technologies, software components, and hardware takes significant time and effort. Any enterprise that values its IT systems and any independent software vendor that wants to make sure its applications work with the platform will gladly choose the certified solution that fits their needs. (Yes, I know, there are a significant number of organizations that choose un-certified goods. More on them later.) As most IT folks know, the cost of software acquisition, proprietary or open source, is far less than the total cost of operation (TCO) over time. Thus, if paying some more upfront means reducing the TCO over time, that's a trade any CIO will gladly make.

(At the risk of losing readers here, the accounting calculations around software amortization are in the favor of upfront costs and then deducting the declining value of the software over time as a "loss," whereas paying for continuing services over the years

will track wage increases over time and cannot be amortized. Thus, some upfront costs with lower TCO makes sense for the bean counters out there, which is a point in favor of purchasing open source products, not necessarily services. Unfortunately, annual renewal fees may muddy this calculation, although I bet that the TCO aspect is still in favor of products, not services, even with annual renewals. Can you tell I married a CPA?)

So how do you know if you're using the particular brand of open source software that is certified for your infrastructure? After all, if it's open source code, then anyone can change it at any time, and you never know what you're getting, right? Actually, that's not true at all, and is one of the great myths of open source software. This is where copyright and trademark law come into play, and any smart open source vendor knows how to leverage the tools of intellectual property law to their advantage.

Let's imagine you have open source project "foo" and you want to transform your highly successful open source project into a software business. You've looked at building services around support and customization, but frankly you have higher ambitions than a mom-and-pop software services and support business. No, you want the big enchilada and are obsessed with changing the world. How do you sell a commercial version of this without resorting to the open core or hybrid approaches, because of their inherent deficiencies?

If you take "foo," devote many man-hours to polishing the software, and then create this splendid unicorn, how do you sell it in a way that connotes its value above and beyond the open source project? Do you call it "foo supreme" "foo super" or just "foo enterprise?" Think about this very, very carefully. You've spent much time, resources and money making "foo" into something most enterprises can use. If you call the product something that evokes the name "foo," what are you saying? Someone might get the impression that it's really just "foo" with some extra naming thrown in but not much else. Does the name "foo ___" confer the effort undertaken to make sure it works cleanly? This has been the great undoing for many open source vendors, from MySQL Enterprise to Hyperic Enterprise Edition and on and on. It also leads to the impression that just plain "Foo" is somehow crippled or otherwise less valued. Remember, you want your open source project and community to be highly valued, otherwise you lose the benefit of an open source strategy and fall back to the problems with open core.

Imagine an alternative scenario. You know that "Foo" is world class software, and you have a really large user and developer base. Now, instead of calling it "Foo X" let's just call it "Bar." What now? By naming it "Bar" you've now created an entire namespace reserved for your commercial efforts. There's no more confusion in the market around what is paid for and what is free. The user and developer communities that either aren't

ready to buy a product yet or never will know that “Foo” is a dynamic open source community with a product that will satisfy their needs. Your prospective customers, the guys who just want something that works, know that they want to check out “Bar” because that’s the thing they’re looking for.

Every day, thousands upon thousands of enterprises are just looking for a product that works - why not give it to them? Perhaps they don’t care if it’s open source, because they’re simply looking for value from their software vendors. They won’t get confused by the name “Foo” because it’s either outside of their day-to-day work, or if they do know about it, they want the certified thing that they know provides a reasonable guarantee about how well it will work in practice. It also helps that the only way to get “Bar” - the certified thing that’s been tested against a wide array of different technologies - can only be obtained through entering into a commercial contract.

This is where protecting your trademark is essential. You are the one who creates the strategy for “Bar,” you’re the one who invests hours into quality assurance and testing, and you’re the one who’s taking on the risk by making a substantial investment in the product. Therefore, you are the only one who gets to create a product called “Bar,” assuming, of course, that you took the time to register the mark in the first place. It’s equally important to protect the mark of the open source platform, Foo. The last thing you want is some other company claiming that their version of “Foo” is “just like ‘Bar”! Thus, it’s important that whoever controls the mark for “Foo,” whether it’s a software vendor or a vendor-neutral organization, also engages in vigorous defense of the trademarks.

SECRETS TO OPEN SOURCE PRODUCTS

Creating, marketing and selling a product is no different in the open source space from any other endeavor. What trips up many software vendors is that they *think* it’s much different, and because of that are led down a path of landmines and less successful strategies.

Considering the previous section about trademarks and namespace, is it really just about the name? Well, yes and no. Calling a product by a different name can be an important step that leads you in the right direction. After the decision to name your product something else from the open source platform, you invariably face a cascade of ramifications that force you to think about product development in a more positive manner, instead of simply thinking defensively about how to prevent your open source project from becoming “too successful.” After naming a product, you need to develop tools specific to your product name, including branding, documentation, etc. You’ll need to facilitate a

community around your product, because you don't want to give your customer community the idea that all they need to do is consult with your open source users. (While helpful, you need your customers to buy into the notion that you, not the volunteer community, are the final word on your product.).

And then you face the decision of what to do with the code itself. If you're taking code from an open source project, don't you have to release that code as open source? In many cases, yes, but that shouldn't affect your product strategy. By all means, *do* release all of your source code, especially the bugs you find from your top-class QA and certification infrastructure. While you may release all of your source code, that doesn't mean you have to release your full test plans, or your QA strategies, or your particular continuous integration suite, or anything else.

And that is the big secret of open source: It's about way more than the code. In order to build a certified, predictable, manageable product that "just works," it requires a lot more effort than just writing good code, although that is the starting point. You're not just testing your code, rather you are testing how well your code integrates with the vast array of components that enterprises are forced to face down on a daily basis. You can release all the code you want, and you don't have to worry about competitors "stealing" your productization process. Some may try, but if they're capable of that, then frankly, they're probably better equipped to sell an open source product in the first place (although nobody likes to hear that).

In the next chapter, I'll go into more detail about the process of differentiating your product from the project(s) you rely on, and how users (freeloaders) of the open source code are actually an essential part of your product-building strategy.

Chapter 4
The New, Open Platform Model

I've been saying for some time now that open source was not about innovation, but rather freedom. It was the freedom to deploy what you want, when you want it that led to the massive global adoption of open source platforms. I get more than a little peeved, then, when I still see references to the open source community in mainstream media circles that imply there's no money in free software. This is what economist Paul Krugman would call a "zombie lie" - an argument that just won't go away no matter how many times you kill it with facts.

When you add up all of the software vendors, service providers, consultants, systems integrators, enterprise IT veterans, and everyone else who works with free software, you're talking about a total ecosystem that measures in the TRILLIONS of dollars, worldwide. We're talking an economy that, if measured as a nation-state, either comes close to or eclipses that of the United States GDP. The sheer amount of financial transactions, internet business, government agencies, archives and every type of industry dependent on free software is staggering.

And yet, we still hear about how open source's success is all just about price - something that couldn't be sold because no one would pay for it. In my talks about open source innovation, I always like to point out that open source and free software are not *just* about price, although that is certainly influential. The reason we know it's not just about price is that we can easily compare results with a well-known product marketing tactic, the "freemium" model. The freemium model is all about giving away something of limited value to be consumed and, hopefully, lead the user to buy the full product in the future. There is certainly a place in the world for freemium products, as Splunk has shown, but freemium didn't win the day - open source did. When you do any basic analysis of customers and how they acquire solutions, it's easy to see why: as a potential customer, why on earth would I ever handcuff myself to a single vendor when I'm not

even sure I like your product? And to invest any time in a freemium product means necessarily losing all of that effort if I then decide to go on to something else. It's much better to spend time and effort that can be recouped should I decide to go with something else. It's this value - freedom - that makes open source software, even open source software you pay money for, inherently more valuable and cheaper over the long-run than freemium products. Thus, we know that the act of open sourcing your software carries with it inherent value for the end user or developer.

This is what the open core folks completely miss - they see open source as just another freemium distribution tool, when it's so much more than that. By devaluing the open source platform, they devalue the very thing that customers actually want. They think about open source solely as a distribution method or, worse, as a way to get free labor so that others will perform the mundane labor of "whitewashing our fence." Word to the wise: NO ONE is going to whitewash your stupid fence, okay? You don't get value in return without first giving up some control.

Now that we've settled that, let's get on with making the sausage, shall we?

SOURCE CODE NUTS AND BOLTS

You may be thinking to yourself, it's all well and good that free software has inherent value, yada yada yada, but how can I use that value to profit handsomely? As we turn our eyes to the making of the sausage, we know that the making of the sausage is never pretty, even if we do rather fancy the final product. Making open source products is no different. I've written the following as a suggestion of how to do it, not necessarily the only, or even best, way.

Now that we've investigated the idea of productization (see Chapter 3), how does it impact engineering practices? For one thing, if you're creating a product with full test suites and certification tests, you probably can't simply use the same release branch as the open source platform. This is another mistake many companies frequently make - they think that productization is as simple as packaging up the open source bits and, presto-change-o, let that money maker roll! The road to hell is littered with the carcasses of a thousand companies that thought productization was just a few sexy demos and some packaging of the bits.
Productization of open source software requires a bit of process hacking and, yes, complexity. First, a diagram of how many open source projects release code:

Open Source Engineering Flowchart

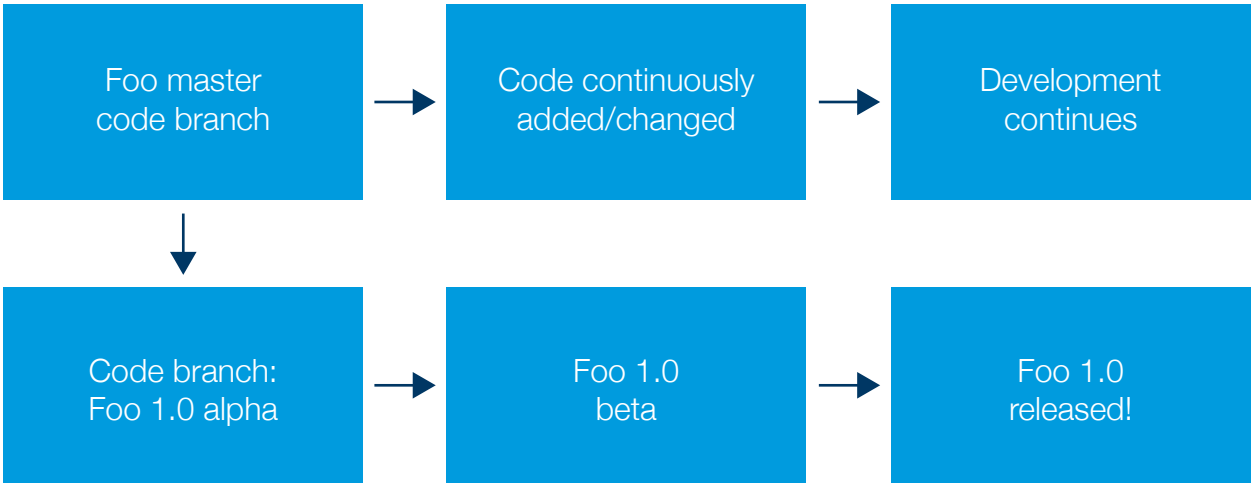


Figure 2: How many open source projects release code.

The open source process will be continuously producing new versions with multiple branches of code. You, the product vendor, must make some interesting choices:

How do you decide which branch(es) to use for your product?
How do you decide what to do with the source code changes you make?
OMG, aren't you making it easier for people to take your hard work and use it free of charge????

The answers, in order:

It depends.
You commit your changes, bug fixes and improvements to their respective upstream branches.
Maybe.

Again, the object here is not to cripple the upstream open source project. Doing so would badly damage your own product, seeing as how you depend on the open source code. You also don't want to maintain your source code changes privately, carrying them forward in your locked source repository, however valuable they might be. You'll end up with a continuous code fork that's never merged back into the open source code, and you'll be maintaining that code fork for all eternity.

You may think that this is no big deal, but rest assured, the last thing you want to saddle your development and product management teams with is an inordinate amount of technical debt that grows substantially over time. You made the decision to create a product from an open source project, now deal with it. That means working to simultaneously improve the open source project in addition to your product. The good news is that by making the decision to occupy a separate product namespace, you no longer have to "prove" which version is the product that you're selling. You're free to continue to work on both project and product, with both evolving, without cannibalizing your product efforts. Remember, you're selling a product that provides value for your customers, not source code. While source code *does* provide some value, it's not the only thing you're selling.

Once you have the engineering processes all worked out, it should look a bit like this:

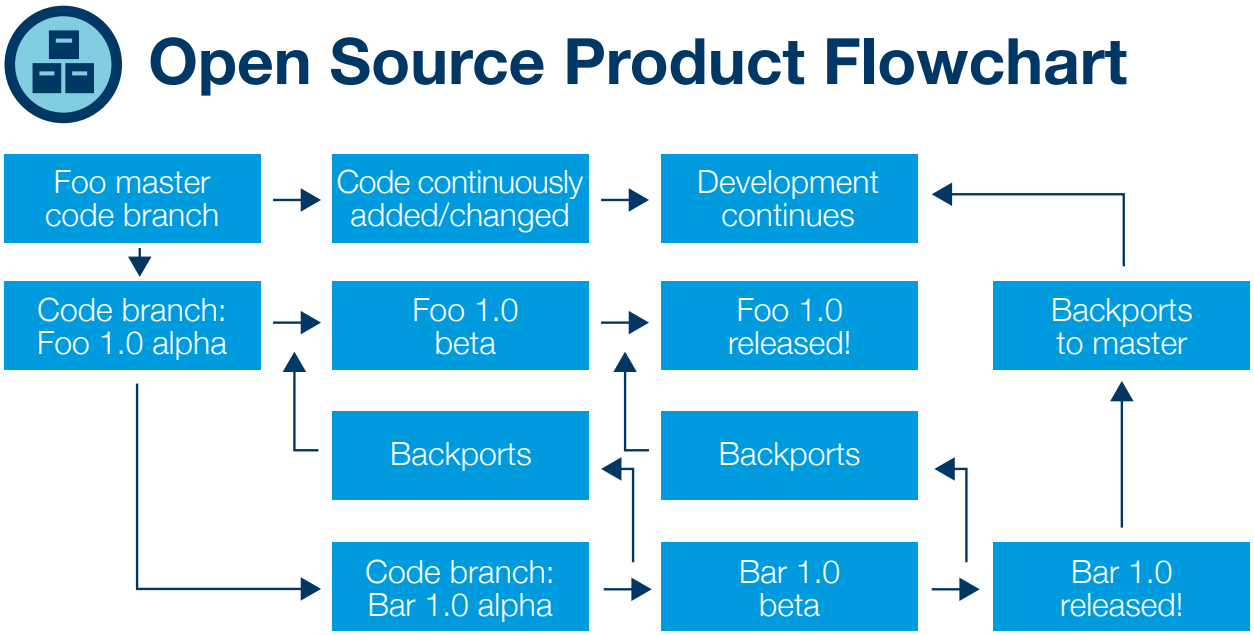


Figure 3: The decision to create a product from an open source project means working to simultaneously improve the open source project in addition to your product.

The key point: "Bar" remains a stable branch that you do all of you testing, certification, QA/QE, packaging, product marketing, etc. It remains relatively static, especially in comparison to the upstream branches. This is a feature, not a bug. The open source releases will probably function similarly to your product branch, but without the QA/QE testing and other hardening that you've baked into your productization process, there's no

guarantee that it will work in all the environments you sell into. Again, it's about the *process* more than just the code. I'll leave it as an exercise for the reader as to what your specific process should be. You're selling a product that is certified against many other software products and hardware configurations.

To further visualize this, forget the source code branches for a bit and imagine the platform as a wholly contained thing:

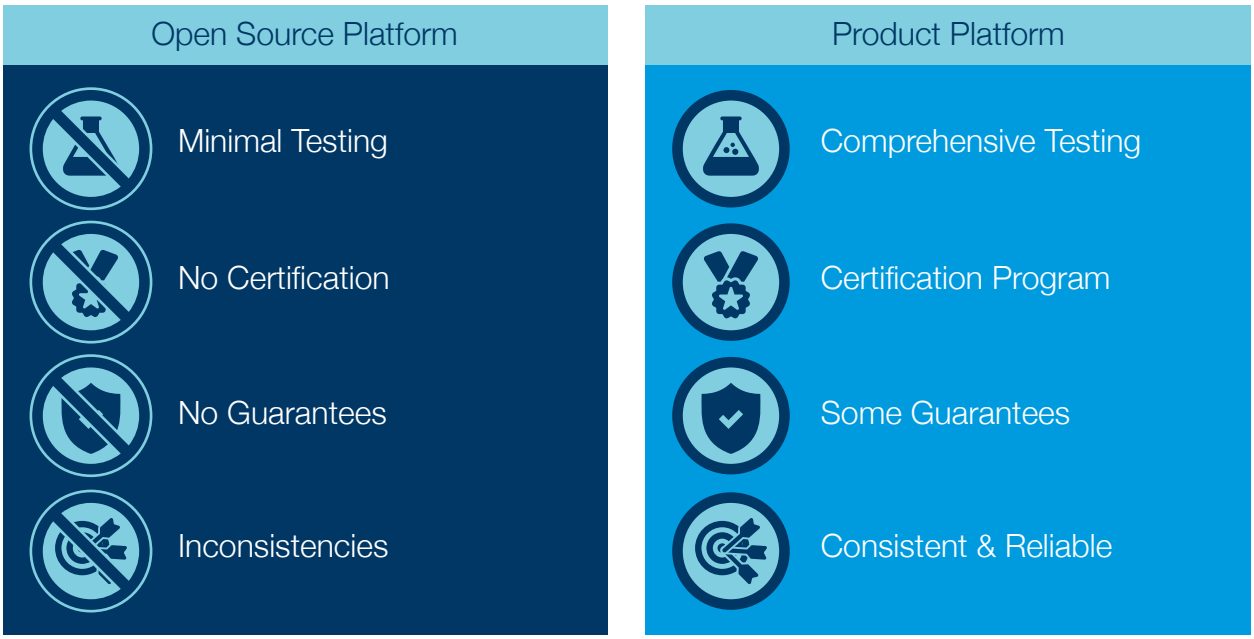


Figure 4: Every day, thousands upon thousands of enterprises are just looking for a product that works - why not give it to them?

With that in mind, your customers are rewarded because the product experience will be highly consistent, stable and not shift under their feet, especially between releases. Your ISVs and other external developers that rely on your product will also appreciate the fact that the API won't be undergoing sudden switches, and the testing suites will help to maintain that level of consistency. And you, the vendor, will be rewarded, because it's much easier to provide product support for something that doesn't change much over time, until of course, it's time to release a new version.

The advantages are obvious, after you start down this path. You're still building and maintaining your open source code, and you're creating a place for customers to come and benefit from all the open source contributions you've made. One anecdote to share here: Back in the day, a certain software vendor split its community product from its

enterprise product. At the time, it was a controversial move, but as one salesperson who was there related to me, the "split" allowed her to focus on selling products to customers who wanted to buy, instead of spending time debating the merits of the GPL with people who were never going to buy anything. Everyone won: The market was less confused about what was sold and what was free, the company won because its product became a runaway success, customers won because they benefited from open source innovation, and the community won because they got a great free product that was produced in the open source way with transparent governance.

PROPRIETARY ADD-ONS

By now, you may think that I, as a free software bigot, would never endorse a proprietary strategy around free software. After all, that's why I dislike open core, right? Actually, I dislike open core because it doesn't work, not because it's ideologically "impure." As I mentioned, what I wrote above is not the only way to write and sell software, but it gives you a good start. Just to prove a point, I'm going to show you a way to utilize open source methodologies while selling proprietary software.

"But wait!" I hear you asking, "Didn't you just say that the open core and hybrid approaches were unsuccessful? Why would we do that?" I'm going to advocate that you start from the position of the valuable open source platform as product and then build on top of that, instead of ignoring the value of the platform as one would do in open core. In other words, the core revenue driver is still the open source platform, and whatever proprietary bits you put together to sell with it serve to create a positive feedback loop, where open source platform adoption creates more potential value for your proprietary add-ons, and the more successful your proprietary add-ons, the more the open source platform becomes a larger center of gravity for its ecosystem. Of course, in this scenario, your add-ons could also be open source - after all, I just showed you how to make an open source product, right?

If you begin with the premise that open source platforms have great value, and you sell that value in the form of a certified software product, that's just a starting point. The key is that you're selling a certified version of an open source platform and from there, it's up to you how to structure your product approach. If, in addition to selling the open source platform, you still want to sell proprietary applications, that's entirely up to you, as long as you don't choose an approach that devalues the platform itself - that is the key underlying difference between successful and unsuccessful approaches. Remember, successful open source platforms are very, very valuable, and enterprises appreciate (and pay for) that value.

WHAT ACTUALLY WORKS

This is not an easy process, and I would never tout it as such, but no successful product process ever is. Creating a successful product is always hard, whether open source or proprietary, but at least after reading this book, I hope you have a better idea of how to take advantage of open source innovation and processes. At least this way, you'll be using methodologies that are known to work.

The key is to make it crystal clear to your audience and market what you're selling (and not selling), anticipate and remove potential confusion, and make sure your target audience understands the inherent value of your open source product. The open source platform and mixed approaches outlined above allow you, the vendor, to provide clarity, which allows your customers and community members to self-select, and makes you, and the wider ecosystem, more efficient.

If you'll notice, the methodologies and strategies I've outlined above apply equally well to internal product development, designed for internal usage at your organization, and for external use by a paying customer. It doesn't matter who the customer or end user is, whether they work for the same company as you or pay for your product as a customer. In Part II, I'm going to turn from processes for internal product development toward a discussion of external processes that affect your ability to deliver good products. We'll cover how to influence open source code and utilize it for higher efficiency. In an open source world, we may depend on outside sources for software, but we have the ability to participate in those upstream sources to ratchet up the potential innovation and fully utilize the resources present in external communities. Put another way, let's talk about supply chain management.

Part 2

Advanced Open Source Product Management

This chapter is adapted and republished with permission from OpenSource.com at https://opensource.com/article/16/12/open-source-software-supply-chain.

Chapter 5
Improving Product Creation & Management

In the previous chapters, I discussed how to take open source software, crank up your product management engine, and produce shiny, polished products for use by others. But what about all that great open source software? Where does it come from, and how can you rely on it? How do you know if an upstream software project is capable of filling your needs? And how do you manage the risks that come with incorporating open source software into your products? This is where we turn to managing your software supply chain, and open source ecosystems introduce some tricky challenges that you'll need to master.

Grasping the nuances of hardware supply chains and their management is straightforward—you essentially are tracking moving boxes. Managing something as esoteric as resources for building software with a variety of contributions made by the open source community is more amorphous.

I used to think of the supply chain for open source platforms as a single process, taking existing open source components and producing a single result, namely a product. Since then, I've begun to realize that supply chain management defines much of the open source ecosystem today. That is, those who know how to manage and influence the supply chain have a competitive advantage over those who don't do it as well, or even grasp what it is.

THE OPEN SOURCE SOFTWARE SUPPLY CHAIN

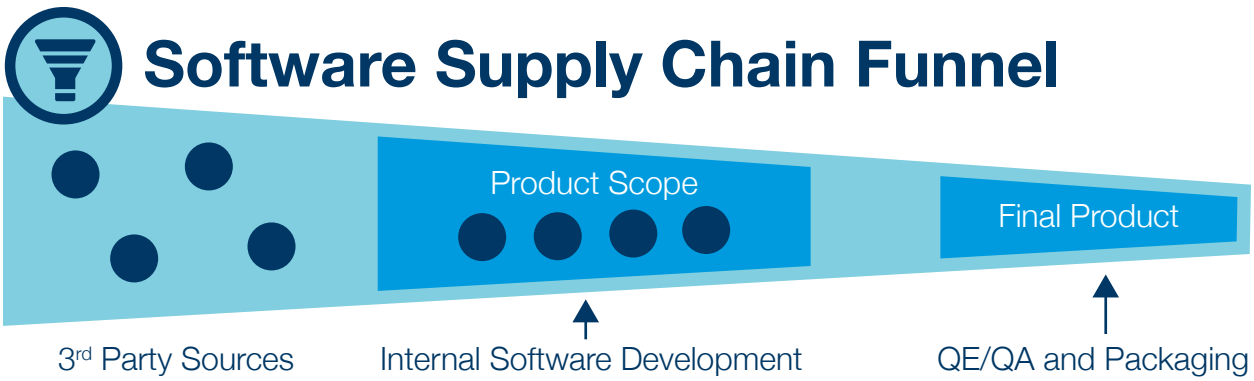
In the hardware world, supply chains and component sourcing is a necessary part of the business. A well-managed supply chain is crucial to business success. How do you determine ideal pricing, build relationships with the right manufacturers, and maximize the efficiency of your supply chain so you're able to produce more products cheaply and

sell more of them? Apple's supply chain is legendary—they were one of the first to realize that a key to competing is building out an effective supply chain of parts, and they did it at a time when Silicon Valley preferred to build most of the components themselves.

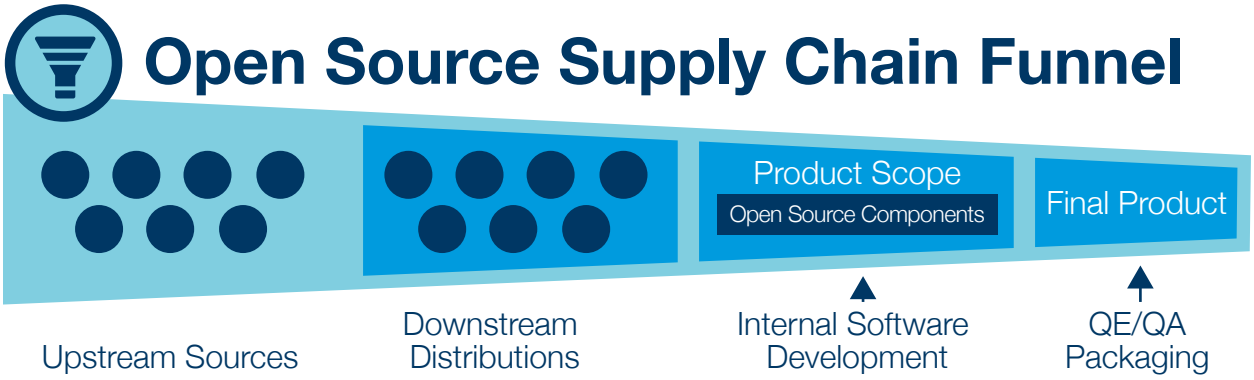
The computing industry laughed at Apple, calling them a "marketing company" that just took parts from elsewhere. These days, every hardware company has an extensive supply chain and dedicates teams of people to managing different aspects of it. The irony is that Apple is probably more innovative on supply chain logistics than they are on actual product. To put it another way, their innovation in supply chain logistics gave them a great platform from which to launch innovative products.

In software, the supply chain has traditionally been much simpler than that for hardware. Whereas hardware supply chains source parts from many different partners in different geographies, traditional software supply chains have mostly defined the process of creating software made in-house, with some third-party software from vendors via commercial license agreements. In this model, most of the supply chain is defined by software sources inside the company itself, possibly from multiple engineering teams. With a small percentage of software coming from outside the company, the process of defining a supply chain was mostly left to internal product management and engineering teams. Yes, licensing software from third-party vendors required license compliance checks as the product was assembled, which meant getting legal approval for any software license agreement. This process was well-established with common practices, with many of the license agreements derived from the same legal template that legal and product management teams had experience with—but then open source became a lot more common, and everything changed.

Software supply chain management went from a relatively simple, well-defined process, like this:



To a chaotic, multi-layered mix of unproven licenses, untested software repositories, and a Wild West mentality that software supply chain teams were ill-equipped to manage. The funnel is now shaped more like:



As you can see, at least one extra layer of complexity has now been added.

You might think that this wouldn't be the case—that simply plugging in open source components would be a direct 1:1 replacement for the traditional method of licensing agreements from third parties, but there's an extra wrinkle to take under consideration. With regards to upstream open source components, many of these raw source repositories have no mechanism for commercial support. As the supply chain or product manager you have no single "throat to choke" when things go wrong, as they inevitably do.

ROLE OF THE SOFTWARE SUPPLIER

You have two basic choices: either build your own internal means of vetting the code and applying product management processes, or rely on an intermediary to perform that function. You can make an argument for creating the processes for pulling down source code, determining legal compliance, applying patches, and getting it ready for production yourself, but it is expensive from a human resources point of view. You should base your decision of whether or not to self-direct the process on its strategic importance to the company and some ROI analysis: If you build a team to manage that process for some software components, will you see a sufficient return on that investment?

In many cases, companies make the decision to go with an intermediary to vet the code, perform some quality assurance engineering, and apply whatever glue code is necessary to make it work satisfactorily. This is where software distributions come into play, filled by companies like Red Hat, SUSE, and Canonical. People often ask me why these

companies are essential, and I hope you can see now why that is the case—because without them, the open source supply chain falls flat.

Without distributions such as Red Hat Enterprise Linux operating system (RHEL) or its equivalent, companies creating products for either internal or external consumption would have to create from whole cloth the processes for pulling in these components, vetting them, hiring the in-house expertise to enhance them, and then perform the gluing process that allows a company to push a release into production. In this scenario, many companies determine that simply letting a distribution company fill in that layer and manage that part of the process is easier and more efficient.

Besides, that's just in the case of the open source "user," not supplier. Here's where it gets really interesting. At what point does a company that uses or inserts open source code into its products decide that it wants to become an influencer or supplier in the supply chain, and what's the best way to do that? One potential conclusion is that to be successful at open source products, you must master the ability to influence and manage the sundry supply chains that ultimately come together in the product creation process. Once done, the end product you produce is able to benefit from your participation on the left side of the supply chain funnel directly.

To be successful at open source products, you must master the ability to influence and manage the sundry supply chains.

ACHIEVING MAXIMUM EFFICIENCY

Instead of maintaining a standard set of patches that you continuously apply to every new vetted upstream component, why not contribute those into the upstream components, making them more easily maintained outside of your organization? If you've already made the decision that working with downstream software distributions is easier than sourcing and vetting the source code yourself, isn't there a direct benefit from working with the distribution vendor to get your code contributed upstream? Doing so results in the added benefit of other groups managing and supporting the maintenance of this code, freeing

up your engineers to work on the interesting stuff that directly adds value to your product. Whether you sell software, sell software consulting services, or create an open source community, studying your supply chain and learning the best way to manage it is worth your time.

Hardware supply chains, based on physical materials, are more static in nature compared to software supply chains. The open source software supply chain is by definition very fluid. Projects wax and wane over time, and you, whether business person, developer, or community leader, must decide which pieces are worth your time and when it's best to cut your losses and switch out one supply chain for another. Which components you use, modify, and create from scratch will all depend on the state of the supply chain that make up your project or product. The process is both proactive and reactive. You must decide proactively which supply chains to invest in and which to ignore, while also reacting to rapid changes in relied-upon ecosystems. Those that master this art will, in theory, have the most efficient processes and, like Apple, will win out in the end. The desired result is achieving a level of efficiency that gives your project team the means to innovate more.

You must decide proactively which supply chains to invest in and which to ignore, while also reacting to rapid changes in relied-upon ecosystems.

If you can efficiently manage your software component supply chains and simultaneously create an efficient supply chain funnel that allows for fast iteration on a product under development, your product creation and management processes will improve. This management requires the investment of resources in not just your product's Quality Engineering (QE) team and supply chain funnel, but also in the supply chains that form the components you use to create your product.

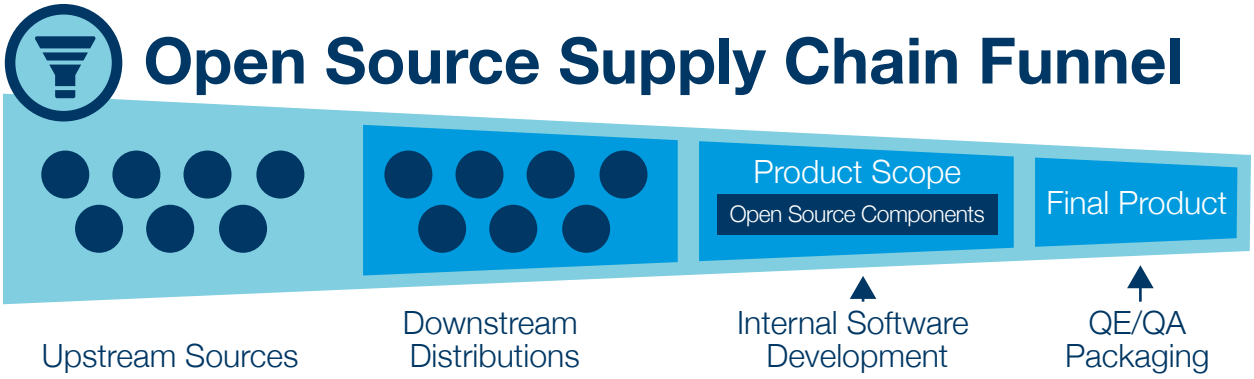
This chapter is republished with permission from OpenSource.com at https://opensource.com/article/17/1/be-open-source-supply-chain.

Chapter 6

Becoming a Supply Chain Influencer

I would bet that whoever is best at managing and influencing the open source supply chain will be best positioned to create the most innovative products. In this final chapter, I'll explain why you should be a supply chain influencer, and how your organization can be an active participant in your supply chain.

In my previous chapter I discussed the basics of supply chain management, and where open source fits in this model. I left readers with this illustration of the model:



The question to ask your employer and team(s) is: How do we best take advantage of this? After all, if Apple can set the stage for its dominance by creating a better hardware supply chain, then surely one can do the same with software supply chains.

EVALUATING SUPPLY CHAINS

Having worked with developers and product teams in many companies, I learned that the process for selecting components that go into a product is haphazard. Sometimes

there is an official bake-off of one or two components against each other, but the developers often choose to work with a product based on "feel". When determining the best components, you must evaluate based on those projects' longevity, stage of development, and enough other metrics to form the basis of a "build vs. buy" decision. Number of users, interested parties, commercial activity, involvement of development team in support, and so on are a few considerations in the decision-making process.

Over time, technology and business needs change, and in the world of open source software, even more so. Not only must an engineering and product team be able to select the best component at that time, they must also be able to switch it out for something else when the time comes—for example, when the community managing the old component moves on, or when a new component with better features emerges.

WHAT NOT TO DO

When evaluating supply chain components, teams are prone to make a number of mistakes, including these common ones:

- **Not Invented Here (NIH)**
  I can't tell you how many times engineering teams decided to "fix" shortcomings in existing supply chain components by deciding to write it themselves. I won't say "never ever do that," but I will warn that if you take on the responsibility of writing an infrastructure component, understand that you're chucking away all the advantages of the open source supply chain—namely upstream testing and upstream engineering—and deciding to take on those tasks, immediately saddling your team (and your product) with technical debt that will only grow over time. You're making the choice to be less efficient, and you had better have a compelling reason for doing so.

- **Carrying patches forward**
  Any open source-savvy team understands the value of contributing patches to their respective upstream projects. When doing so, contributed code goes through that project's automated testing procedures, which, when combined with your own team's existing testing infrastructure, makes for a more hardened end product. Unfortunately, not all teams are open source-savvy. Sometimes these teams are faced with onerous legal requirements that deter them from seeking permission to contribute fixes upstream. In that case, encourage (i.e., nag) your manager to get blanket legal approval for such things, because the alternative is

carrying all those changes forward, incurring significant technical debt, and applying patches until the day your project (or you) dies.

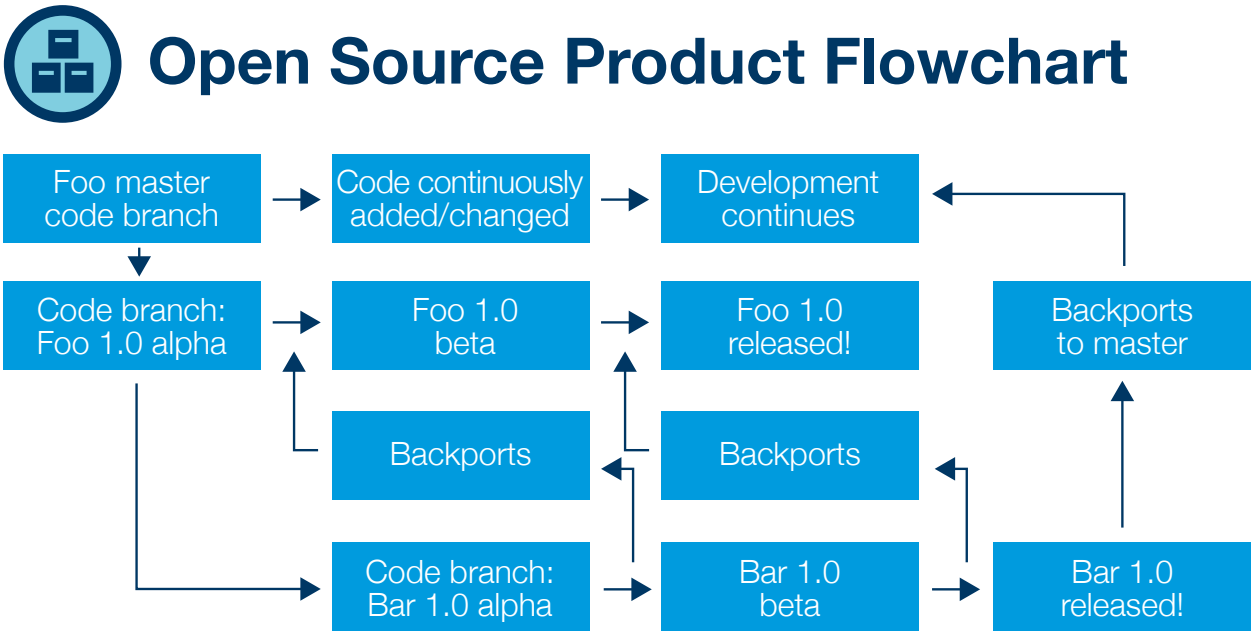
- **Think you're only a user**
  Using open source components as part of your software supply chain is only the first step. To reap the rewards of open source supply chains, you must dive in and be an influencer. (More on that shortly.)

EFFECTIVE SUPPLY CHAIN MANAGEMENT EXAMPLE: RED HAT

Because of its upstream-first policies, Red Hat is an example of how both to utilize and influence software supply chains. To understand the Red Hat model, you must view their products through a supply chain perspective.

Products supported by Red Hat are composed of open source components often vetted by multiple upstream communities, and changes made to these components are pushed to their respective upstream projects, often before they land in a supported product from Red Hat. The workflow looks somewhat like:

There are multiple reasons for this kind of workflow:



- **Testing, testing, testing**
  By offloading some initial testing, a company like Red Hat benefits from both the upstream community's testing, as well as the testing done by other ecosystem participants, including competitors.

- **Upstream viability**
  The Red Hat model only works as long as upstream suppliers are viable and self-sustaining. Thus, it's in Red Hat's interest to make sure those communities stay healthy.

- **Engineering efficiency**
  Because Red Hat offloads common tasks to upstream communities, their engineers spend more time adding value to products for customers.

To understand the Red Hat approach to supply chain, let's look at their approach to product development with OpenStack.

Curiously, Red Hat's start with OpenStack was not to create a product or even to announce one; rather, they started pushing engineering resources into strategic projects in OpenStack (starting with Nova, Keystone, and Cinder). This list grew to include several other projects in the OpenStack community. A more traditional product management executive might look at this approach and think, "Why on earth would we contribute so much engineering to something that isn't established and has no product? Why are we giving our competitors our work for free?"

Instead, here is the open source supply chain thought process:

- **Step 1**
  Look at growth areas in the business or largest product gaps that need filling. Is there an open source community that fits a strategic gap? Or can we build a new project from scratch to do the same? In this case, Red Hat looked at the OpenStack community and eventually determined that it would fill a gap in the product portfolio.

- **Step 2**
  Gradually turn up the dial on engineering resources. This does a couple of things. First, it helps the engineering team get a sense of the respective projects' prospects for success. If prospects aren't good, the company can stop contributing, with minimal investment spent. Once the project is determined to be worth the investment, the

company can ensure its engineers will influence current and future development. This helps the project with quality code development, and ensures that the code meets future product requirements and acceptance criteria. Red Hat spent a lot of time slinging code in OpenStack repositories before ever announcing an OpenStack product, much less releasing one. But this was a fraction of the investment that would have been made if the company had developed an IaaS product from scratch.

- **Step 3**
  Once the engineering investments begin, start a product management roadmap and marketing release plan. Once the code reaches a minimum level of quality, fork the upstream repository and start working on product-specific code. Bug fixes are pushed upstream to openstack.org and into product branches. (Remember: Red Hat's model depends on upstream viability, so it makes no sense not to push fixes upstream.)

Lather, rinse, repeat. This is how you manage an open source software supply chain.

DON'T ACCUMULATE TECHNICAL DEBT

If needed, Red Hat could decide that it would simply depend on upstream code, supply necessary proprietary product glue, and then release that as a product. This is, in fact, what most companies do with upstream open source code; however, this misses a crucial point I made previously. To develop a really great product, being heavily involved in the development process helps. How can an organization make sure that the code base meets its core product criteria if they're not involved in the day-to-day architecture discussions?

To make matters worse, in an effort to protect backwards compatibility and interoperability, many companies fork the upstream code, make changes and don't contribute them upstream, choosing instead to carry them forward internally. That is a big no-no, saddling your engineering team forever with accumulated technical debt that will only grow over time. In that scenario, all the gains made from upstream testing, development and release go away in a whiff of stupidity.

BECOME THE SUPPLY CHAIN

Once you begin to understand Red Hat's approach to supply chain, which you can see manifested in its approach to OpenStack, you can understand its approach to

OpenShift. Red Hat first released OpenShift as a proprietary product that was also open sourced. Everything was homegrown, built by a team that joined Red Hat as part of the Makara acquisition in 2010.

The technology initially suffered from NIH—using its own homegrown clustering and container management technologies, in spite of the recent (at the time) release of new projects: Kubernetes, Mesos, and Docker. What Red Hat did next is a testament to the company's commitment to its open source supply chain model: Between OpenShift versions 2 and 3, developers rewrote it to utilize and take advantage of new developments from the Kubernetes and Docker communities, ditching their NIH approach. By restructuring the project in that way, the company took advantage of economies of scale that resulted from the burgeoning developer communities for both projects.

Instead of Red Hat fashioning a complete QC/QA testing environment for the entire OpenShift stack, they could rely on testing infrastructure supplied by the Docker and Kubernetes communities. Thus, Red Hat contributions to both the Docker and Kubernetes code bases would undergo a few rounds of testing before ever reaching the company's own product branches:

The first round of testing is by the Docker and Kubernetes communities.Further testing is done by ecosystem participants building products on either or both projects. More testing happens on downstream code distributions or products that "embed" both projects. Final testing happens in Red Hat's own product branch.

The amount of upstream (from Red Hat) testing done on the code ensures a level of quality that would be much more expensive for the company to do comprehensively and from scratch. This is the trick to open source supply chain management: Don't just consume upstream code, minimally shimming it into a product. That approach won't give you any of the advantages offered by open source development practices and direct participation for solving your customers' problems.

To get the most benefit from the open source software supply chain, you must be the open source software supply chain.

CONCLUSION

Throughout this ebook, I've walked through both internal as well as external processes and methodologies that will help you in your quest to create better, more resilient, open source products or services. Whether you're a CIO, DevOps professional, product manager, entrepreneur or otherwise tasked with creating open source products, you now have the building blocks on which to build your success. And if you're an investor, you now have the basis on which to judge the product management discipline of companies in your portfolio.

Using the methods in this book, you can now understand how to utilize platforms of innovation and more efficiently build your products with them. You'll know how to separate your innovation bits from the product management bits and how they are related but not the same. And above all, you can now appreciate how source code is not product — although it's a key building block.

I hope you enjoyed this ebook, and I hope to hear all about your adventures on Open Source Entrepreneurs Network, osenetwork.com!