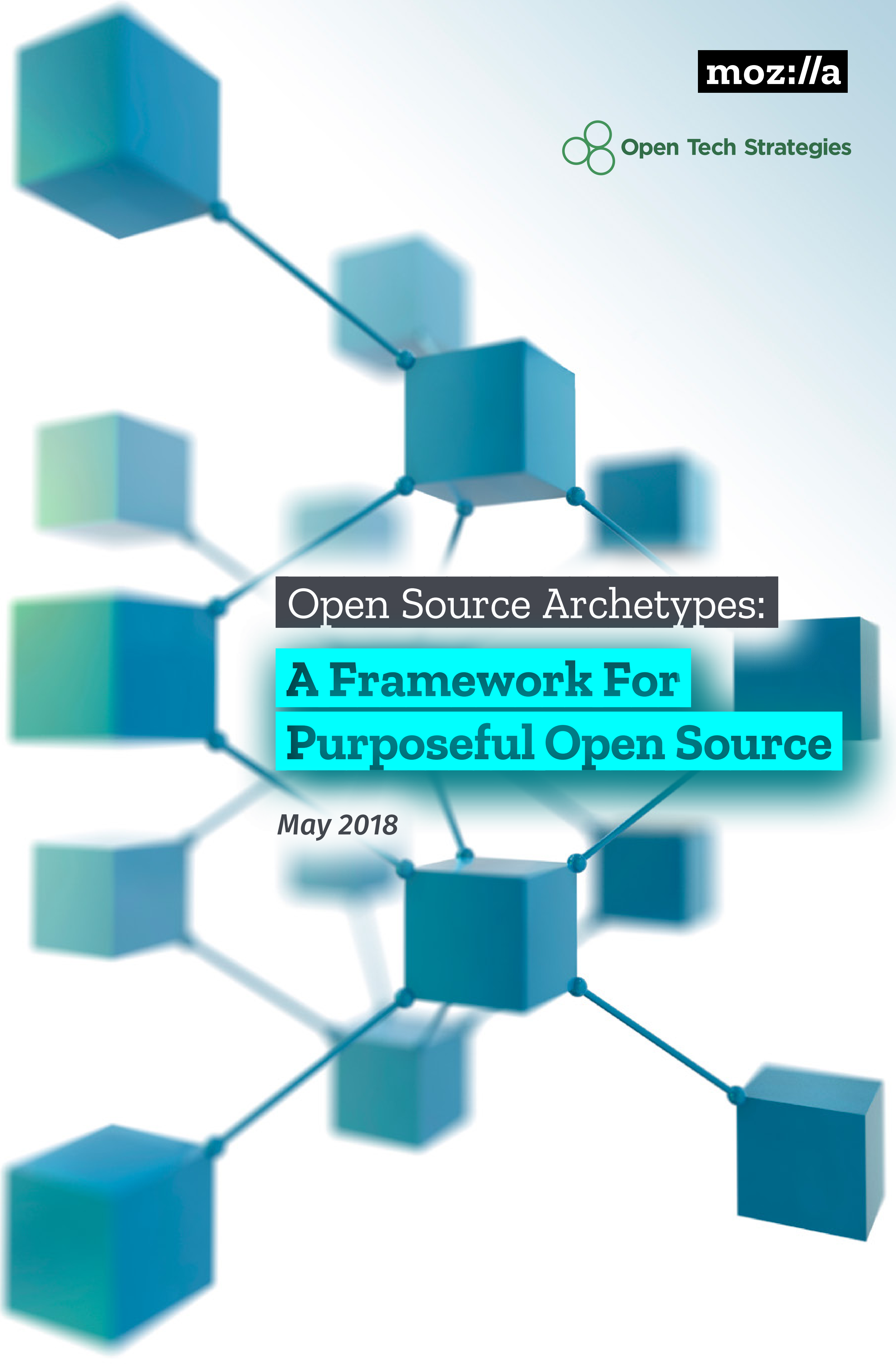
moz://a

Open Tech Strategies



Open Source Archetypes: A Framework For Purposeful Open Source

May 2018

Table of Contents

Preface

This report was originally commissioned by Mozilla for internal purposes. Mozilla wanted a shared framework for discussing its options in running open source projects, and requested a survey and comparative analysis of open source project archetypes — the various shapes that open source projects take in order to meet their various goals. For example, should a project prioritize development momentum over early collaborator acquisition, or ***vice versa****? Should it optimize for individual contributors or for institutional partners? Different projects will have different answers to these and many other questions.*

The benefits of having a common vocabulary of archetypes extend beyond Mozilla, however, and the vocabulary is more useful and more easily improvable the more widely it is shared. Accordingly, Mozilla decided to release the report publicly. Very little has been changed from the original internal version: the Mozilla-specific orientation has been left intact, on the theory that the analysis will be clearer if tuned to a specific (and fortunately well-known) organization rather than rewritten for a hypothetical general audience.

We emphasize that this is very much a version 1.0. There may be important archetypes we have not thought of, and for the archetypes we have identified there may be important properties not discussed here. We intend this report to be the beginning of a longer conversation about open source project archetypes. Version 2.0 may or may not come from us, and may or may not even come from a single source; we think it will exist eventually, however, and we hope that when it does 1.0 will be regarded as a worthy ancestor.

Introduction

Producing open source software is part of the core culture at Mozilla, and has been since the organization's founding. It is one of Mozilla's greatest strengths and is a key element of Mozilla's identity both internally and in the world at large.

By "open source", we mean software released publicly under a license that is both a free software license according to the FSF[1] and an open source license according to the OSI[2] [3]. But this definition really only describes a mode of software ***distribution****; it does not imply any particular mode of* ***production****. While open source distribution is built into Mozilla's identity and underpins the organization's mission, just saying "open source" actually provides little guidance as to how Mozilla should manage its projects or its relations with partner organizations, developers, and users in order to gain strategic market advantages.*

The purpose of this report is to make it easier for Mozillians to talk productively about open source choices, goals, and tradeoffs. These discussions may be had internally, with organizational partners, with contributor communities, and in some cases with users and with media organizations. Mozilla needs to be able to decide what type of open source project a given project should be in order to best further Mozilla's mission, and the answer may be different for different projects.

The report provides a set of ***open source project archetypes*** *as a common starting point for discussions and decision-making: a conceptual framework that Mozillians can use to talk about the goals and methods appropriate for a given project. The framework is not intended to resolve differences or to rank archetypes and projects from worst to best. Instead, we submit these archetypes as a shared vocabulary and an already-mapped territory that can help Mozilla understand a range of open source approaches and the potential benefits of each one.*

The purpose of this report is ***not*** *to provide a single, shared understanding of open source methodology to be agreed on and adopted across Mozilla. There is no need to unify around just one understanding, and attempting to do so would be harmful, given the diversity of products and communities Mozilla works with. Indeed, the Mozilla way might be a willingness to explore many approaches to open source.*

[1] https://www.fsf.org/ [2] https://opensource.org/

[3] *This definition excludes the Creative Commons Attribution (CC-BY) license, the Attribution-ShareAlike (CC-BY-SA) license, and the CC-Zero (CC0) public domain dedication. Although they are open source in spirit, for various technical reasons they are not good choices for software source code. CC-BY and CC-BY-SA are sometimes appropriate for documentation and other non-code assets, as mentioned in section "Questions to Consider", and we consider a project where the source code is released under an open source software license and the documentation under one of these CC licenses to be an open source project overall. For more discussion of CC0 and how the public domain is not automatically like an open source license, see https://opensource.org/faq#cc-zero and https://opensource.org/faq#public-domain.*

How to Use This Document

The archetypes presented here (categorized in section "Open Source Project Archetypes") are not business strategies in themselves; rather, they are consequences of business strategy decisions.

For any project that Mozilla starts, acquires, or revives, the question of how to run it must start with the question of what Mozilla is trying to accomplish with it in the first place. Once there is a clear understanding of the project's goals, it becomes possible to discuss what kind of an open source project it should be.

We therefore suggest that this report be used as follows:

1. First articulate to all stakeholders the goals of the project: what aspects of Mozilla's mission will it serve, and, from a product perspective, how will it do so? Why has Mozilla undertaken this project? These goals should be based upon a shared understanding of the market ecosystem.
2. Review the array of potential benefits that being open source can bring, as described in "Benefits of Open Source". Which ones, if any, resonate with your project's needs and goals? Note that no project should seek (or be expected to obtain) all of these benefits. The point of the review is to stimulate thought about which benefits are most important for your particular project — and for which you should therefore optimize.
3. Based on an understanding of project goals and the particular open source benefits you want to generate, find the archetype that seems like the best fit (see "Open Source Project Archetypes").
4. Use your chosen archetype to help drive a discussion within the project team and, as merited, with key external stakeholders. Review the questions in "Practical Questions To Ask About A Project", to help determine if the archetype seems accurate, and if not, why not. The answers generated through this discussion will help the team make informed decisions about the open source project's licensing, governance and co-development models, and general management.

The set of archetypes in section "Open Source Project Archetypes" is deliberately coarse-grained and simplified. It is not important that a given project exactly match every aspect of a particular archetype; some projects may even incorporate elements from different archetypes. In general, however, most projects will fit best with one archetype. After that, governance, licensing, process, and infrastructure decisions can then be made in light of that archetype, with some adjustments for the project's specific circumstances.

Benefits of Open Source

In order to decide which particular benefits of open source to optimize for, it helps to survey the reasons a project is open source in the first place. Below is a checklist of high level benefits that being open source can bring; most projects will be motivated by some sub-set of this list. Knowing which benefits are most important for a given project, and their relative priority, will help identify which archetype from section "Open Source Project Archetypes" would best serve that project's goals.

Note that this list of benefits is oriented toward organizations. There will always be individuals who make a personal decision to open source their code; their motivations may include some of the ones below, but may also include personal gratification or philosophy, or even sheer habit. For this report, we focus on organizational and mission benefits that would be relevant to Mozilla.

Open source can be used to:

Lead a standardization effort in a particular area.

For this purpose, widespread adoption usually takes precedence over building a robust contributor ecosystem, at least initially. Licensing that is compatible with being used in proprietary systems may also be important, which would imply a noncopyleft license (see "Basics of Open Source Licensing").

Improve product quality.

This works most effectively in archetypes that have a dedicated core team that priortizes involvement from outside developers. The core team acts as a "quality ratchet", filtering the best ideas from the broad developer pool into the project, ensuring that those ideas get implemented and stay implemented (often the core team devotes proportionally more time to QA and test suite improvements than other developers do), and actively encouraging continued participation so that more total development energy is sustained in the project than the core team could ever muster on its own.

Amplify or expand developer base.

Like many organizations, Mozilla is sometimes in the position of wanting to build something and knowing that they cannot build it alone. Open source sends a strong signal that Mozilla is seeking like-minded collaborators, and provides a framework for collaboration once they've been found.

Which archetypes best suit this goal depends on the nature of the project, but in general they will be archetypes that tend to invest more up-front effort in making the project welcoming and easy for others to gain influence in (as opposed to archetypes that optimize

Benefits of Open Source (continued)

for reaching a minimum viable product early with internal resources, or that optimize for maintaining Mozilla control over the long term).

Increase the size or quality of your organization's developer hiring pool.

Open source engagement puts an organization in much greater contact with potential hires who have expertise in technology the organization values. It also creates a public portfolio of a potential hire's work. This includes not just their code commits but also their communication patterns, collaboration style, and personality. This increased window on a candidate's abilities provides a significant amount of information relevant to identifying candidates and deciding to hire them.

Improve internal developer morale and retention.

Developers want to participate in open source. All of the factors that bring outside developers into a hiring organization's view also make their own developers visible to other firms. Open source allows developers to build a public portfolio and expertise in widely-used technology (as opposed to expertise in a proprietary codebase, which doesn't travel very well). The best developers increasingly demand the ability to display their skills in public ways that create future opportunities. Open source does that in ways proprietary pursuits simply cannot.

How this affects retention depends on the organization. Here Mozilla has an advantage thanks to its mission-driven nature: Mozilla is better able to keep good developers once hired than the typical tech company is, because Mozilla offers significant non-monetary job satisfaction. But the same is true, if to a lesser degree, at any organization that offers a good workplace for developers. Thus open source activity offers a relatively ***greater*** benefit to organizations that are already good places to work: it reminds developers of their market value, which in turn contributes to healthy morale and makes the developers' current position remain attractive.

Open source involvement also allows a developer to exercise a more varied set of skills and interact with a wider group of people. The morale benefit of expanding a job role in this way can be significant.

Improve internal collaboration within one's own organization.

Effective open source is all about removing collaboration barriers. Often people think of those barriers as occurring at entity boundaries, but large organizations usually have a variety of internal boundaries as well. Adopting structures that encourage inter-team collaboration tends to increase it regardless of whether the two teams reside at different companies or simply different business units.

Benefits of Open Source (continued)

Establish a basis for product reputation.

For some people and organizations, a product's being open source is a significant factor in adoption decisions.

Establish a basis for organizational brand and credibility.

Above the level of the individual project, hosting a collection of well-run open source projects can attract partner interest and media interest, and in certain circumstances increase political influence.

Improve market acceptance by eliminating customers' and partners' exposure to the risks inherent in proprietary approaches.

This benefit views open source as a form of insurance, similar to code escrow. But in addition to the escrow of the code itself, the continued investment of other contributors is in a sense escrowed as well, and is in principle available to be retargeted should Mozilla ever decide to shift resources away from the project.

Breadth of developer base — and in particular diversity among the different organizations who supply participating developers — is part of that insurance. Aside from raising the project's own bus-factor[4], broad contributorship means that third parties will have more routes by which to procure support when they need it.

Thus, for this benefit to be fully realized, it is often best to choose an archetype that invests in building a broad developer base in which Mozilla is just one of many participants, or at most "first among equals". Note that this may be in tension with archetypes that are more optimized for making rapid technical progress. If Mozilla dictates direction and provides the majority of the driving energy in the project, potential adopters will realize that and take into account the risk to the project of Mozilla changing its investment level down the road.

Stop someone else from establishing early dominance in an particular area.

Because open source can spread so quickly, often assisted by third-party developers and packagers, a timely release can sometimes gain a significant share of a particular user base against a more slowly-distributed proprietary alternative — even when that alternative is initially ahead in terms of features or user experience.

Engage with users.

Open source can be a very effective way to learn who is using the software, how they are using it, and to draw some of them into closer participation and contribution.

[4] *https://en.wikipedia.org/wiki/Bus_factor*

Benefits of Open Source (continued)

How much these benefits are available depends a great deal on the nature of the software and the demographics of its user base (e.g., there's a reason why so many contributions to TeX and LaTeX are from graduate students in technical fields), but some archetypes are better suited to broad user engagement than others.

For example, a project can put extra effort into making it easy for non-technical users to report bugs or have conversations with support staff and even with developers. This wouldn't necessarily be a good use of resources for all projects, but it could be for a project that prioritizes user engagement, or prioritizes conversion of individual users (as opposed to organizationally-motivated users) into contributors.

In general, the larger the audience for a given piece of software, the less likely it is that the users who engage with it as an open source project will be representative of the user base as a whole. However, the slope of this curve can be influenced by choice of archetype.

Provide a framework for collaborating with partners.

When a project is not open source, bringing partners in as participants often involves cumbersome overhead: negotiating MOUs, signing NDAs, etc. On the other hand, when the project is open source, then organizational partners can just participate using the same structures as everyone else, and Mozilla still has the option to grant preferential attention or priority to the partners as needed. Of course, this works best in a project archetype where the governance and prioritization mechanisms do not impose constraints that would prevent Mozilla from acting on such preferences.

Provide transparency to customers and partners.

Sometimes source code transparency — which is always guaranteed by open source — is enough. But sometimes process transparency is desirable as well. For example, a partner may want to see that their competitors' needs aren't being unduly favored. Process transparency provides reassurance by showing the partner how they too can influence the roadmap of the project to satisfy their needs.[5]

Increase insight into partners' (or competitors') strategy and goals.

One can extrapolate information about partners and competitors from how their representatives behave in an open source project. This is most effective when not just developers but their managers observe the open source project closely; managers may also have a better sense of market context and thus be able to infer more information than developers could from observing external participants' behavior.

[5] *Some common ways to gain such influence in an open source project are to invest more time from current developers, or in some cases to hire a person who is already influential in the project.*

Open Source Project Archetypes

The archetypes described here are simplified versions of those found in the wild.[6] The list aims for broad coverage: ideally, it should be possible to match any real-world open source project to one of these archetypes, with some adjustment for circumstances. Even though certain archetypes won't be appropriate for Mozilla to use for the projects it launches, Mozilla will still often encounter projects with those types — through engineering collaboration, through partnerships, even through purchase — and being able to discuss how best to work with or influence such projects can be useful. However, we will devote less space to archetypes that Mozilla is unlikely ever to use itself.

These archetypes are intended to help Mozilla with internal discussions as well as with public positioning. Externally, it is important for Mozilla to set clear public expectations about every project it starts. This increases others' faith that Mozilla is a good project host and a good partner to work with. For example, if Mozilla openly declares that a certain project is going to be largely funded by and wholly controlled by Mozilla for the next few years, and uses the project's archetype in explaining why, then even potential partners who rely on that declaration in choosing (for now) not to participate will still have had a positive experience with Mozilla. Their belief that they ***could*** partner with Mozilla on something will be reinforced, because Mozilla took care to explain itself and not to waste their time.

It is perfectly fine to launch a project using one archetype while stating a plan to switch to another down the road. We include a brief discussion of the evolution of archetypes for this reason. For example, a project may need to be tightly focused in its early stages, but plan to change to a more distributed style of leadership and decision-making once it attracts a community of diverse interests.

Each of these archetypes is presented with a brief description of its key characteristics, including how the sponsoring organization could best benefit from it. Usually this is followed by an example or two described in some detail, followed by key points on licensing, codevelopment, and community, and how tightly any components of the project are related.

[6] *There might be some important archetypes that we simply didn't think of. One of the fastest ways to discover them is to publish a report claiming to list all the archetypes and then wait for the bug reports to roll in; that is our approach here.*

Business-to-Business (B2B) Open Source

EXAMPLES

Android, Chromium

Characteristics: Solid control by founder and (sometimes grudging) acceptance of unpredictability by redistributors. This also implies a long fork-modify-merge cycle for partners, and thus a constant temptation on their part to fork permanently.

This archetype aims at ubiquity. Its open source nature is designed to spur OEM adoption by partners and competitors across the industry and to drive out competing options in the same way pricing browsers at zero once discouraged new entrants into the browser market. It can also drive down the price of complementary products: for example, as Red Hat Enterprise Linux (RHEL) does for server hardware and Google's Android operating system does for handheld devices.[7]

It is worth noting that Google does not derive much direct revenue from Android. Instead, Android's popularity puts Google's products and search preferences in the hands of users, and that creates opportunities for Google. Android users default to Google's search engine, buy media from Google, pay for apps in Google's app store, provide a river of data for Google to mine, and favor Google's app ecosystem (Calendar, Gmail, Maps, etc). All of that generates revenue and strategic advantage for Google. This archetype is thus a strategy for gaining marketshare as a ***revenue opportunity***.

This model probably works best when run by a fairly large company with multiple ways of applying market pressure and multiple channels for distribution. It is difficult to wield without considerable financial resources and other strategic advantages.

Licensing: Almost always non-copyleft.

Community standards: In general, the lead company does not put much emphasis on welcoming or nurturing contributors; exceptions may be made for strategically important organizational partners.

Component coupling: Tightly coupled modules, to allow the lead company to market one unified product.

Main benefits: Can drive industry adoption of a technology that is strategically important to your organization.

Typical governance: Maintainer-led by a group within the lead company.

[7] *Compare with the Mass Market archetype, which can have similar effects on directly competitive products but is less likely to affect complementary products.*

Multi-Vendor Infrastructure

EXAMPLES

Kubernetes, Open Stack

Characteristics: Multi-Vendor Infrastructure are large software projects that provide fundamental facilities to a wide array of other businesses. This infrastructure is designed collaboratively by multiple vendors and then each deploys it in their own way. Kubernetes and OpenStack are two prominent Multi-Vendor Infrastructure projects that most developers will have heard of.

Multi-Vendor Infrastructure efforts are driven by business needs rather than individuals, and are open to relatively lightweight participation. A good way to distinguish a Multi-Vendor Infrastructure project from Business-to-Business (B2B) Open Source is to look at how often contributors continue to participate after they switch employers. When that phenomenon occurs, it indicates that multiple organizations use the software as infrastructure in roughly the same way, and thus the same person may continue representing those technical priorities across different organizations.

In other words, the difference between this and B2B Open Source is that less organizational and process overhead is required to take part — enough so to make participating a qualitatively different proposition. For example, individual developers are generally more able and more likely to make contributions to Kubernetes and OpenStack than they are to Android, ***but*** they still tend to do so from organizational rather than from individual motivation. These organizational motivations often relate to tactical needs, and may be enabled by mid-level management decisions. By contrast, B2B open source tends to be driven by corporate business strategy, and is thus more likely to figure into executive-level decisions.

What this means for Mozilla is that if Mozilla clearly understands who within its potential partners is authorizing participation, then Mozilla will be much better positioned to tune the project's collaboration processes — and related things like conference appearances, media strategies, further partner discovery, etc. — so as to maximize the benefit to Mozilla's mission.

Note that both the Multi-Vendor Infrastructure and the B2B Open Source archetypes are patterns that achieve benefits at scale. They are also patterns that provide infrastructure to enable a varied set of third-party activity. The largest advantage of open source to these ecosystems is broad standardization (including informal or ***de facto*** standardization). Both the container world and the cloud ecosystem provide a set of tools and pieces that can be combined together in standard ways to provide custom solutions. That standardization is necessary to allow reconfiguration and customization. It drives adoption at scale, and, in a virtuous circle, the scale is what creates the ***de facto*** standards. It would be difficult to achieve this dynamic with a proprietary approach; open source lowers the cost of participation and powers network effects that lead to scale.

Licensing: Typically non-copyleft, given that many members of the community are likely to maintain proprietary forks of the core product.

Community standards: Welcoming, but formal, and often difficult for individual contributors to enter. Participation takes a high level of resources.

Component coupling: Loosely-coupled modules joined by ***de facto*** standards.

Main benefits: Fosters collaboration with partner organizations to solve shared problems.

Typical governance: Formal tech committee, with membership by each of the major contributing companies.

Rocket Ship To Mars

EXAMPLES

Meteor, Signal

Characteristics: "Rocket Ship To Mars" projects are characterized by a small full-time core team that is wholly focused on a well-articulated and highly specific goal. They are often led by charismatic founders and enjoy a funding runway sufficient for bootstrapping. Their open source strategy is often rooted in a commitment to ***transparency*** and providing ***insurance***: they want to instill confidence among developers and users in order to promote adoption, and being open source is one ingredient in doing that.

Rocket Ships To Mars tend toward non-copyleft licensing when their goal is broad adoption by the tech industry at large, but may use copyleft licensing when aiming for deployment and usage scenarios where there is no particular advantages to non-copyleft. (For example, Signal notes that their use of the GPL-3.0 license is primarily for "quality control"[8].) In any event, Rocket Ships To Mars tend not to play well with others. Every ounce of fuel goes to thrusters, which leads little energy left over for the hard work of nurturing a contributor community.

Rocket Ships To Mars typically do not invest a lot of effort in encouraging broad contributorship, not so much because they wouldn't welcome contributions as because it is hard to find contributors who share the project's intensity and specificity of vision, and who are sufficiently aligned with the founders to take the time to make their contributions fit in. These projects are also usually trying to reach a convincing alpha release as quickly as possible, so the tradeoff between roadmap speed and contributor development looks different for them than for other types of projects.

Similarly, the components of such projects are generally tightly coupled: their goal is not to create an ecosystem of plug-and-play modules, but to create one core product that can be marketed as a standalone service.

The biggest challenge for Rocket Ship To Mars projects is usually to detect and take advantage of the right ***organizational*** collaboration opportunities. Some strategic partnerships are worth sacrificing momentum for, especially when they could affect the project's direction early on.

[8] *See https://signal.org/blog/license-update/, where Moxie Marlinspike writes "We like the GPL for the quality control that it provides. If someone publicly says that they're using our software, we want to see if they've made any modifications, and whether they're using it correctly."*

Part of the purpose of specifying this archetype is to help Mozilla give itself permission to run projects like this when appropriate. There are times when Mozilla may want to press forward to some goal and not stop (at least for a while) to deal with other participants.

At the very least, the people advocating for this should have a model to point to, and the people arguing against it should have a clear idea of what it is they're arguing against.

When a Rocket Ship To Mars project is launched, it is important to set public expectations accurately. Some observers will be interested in going to Mars, and some will not be, but at least when there is clear messaging each observer can make an informed decision about how much to invest in following the project or in aligning themselves with it sufficiently to be able to participate on Mozilla's terms.

Licensing: Usually non-copyleft, but may be copyleft under certain circumstances.

Community standards: Difficult to enter; focused on the core group.

Component coupling: Tight, in order to ship one core product.

Main benefits: Achieves a quick, focused effect on a specific area; if successful, can co-opt competition.

Typical governance: Maintainer-led by the founding group.

Controlled Ecosystem

EXAMPLES

WordPress, Drupal

Characteristics: Real community involvement, and diversity of community motivations, but with a shared understanding that the founder (or some credible entity) will act as “benevolent dictator”. Requires experienced open source community management on part of project leaders and occasional willingness to compromise. Works best when technical architecture directly supports out-of-core innovation and customization, such as via a plugin system.

Controlled Ecosystem efforts find much of their value in that ecosystem. The core pro vides base value, but the varied contributions across a healthy plugin ecosystem allow the project to address a much larger and diverse set of needs than any one project could tackle alone. Over time, these projects might see more and more of the core functionality structured as plugins as the product becomes more customizable and pluggable. This increasingly lets the plugins determine the end-user experience, and the core project can eventually become infrastructure.

Licensing: Can be either copyleft or non-copyleft. When copyleft, decisions must be made about whether the core interfaces (and maintainer-promulgated legal interpretations) encourage or discourage proprietary plugins.

Community standards: Welcoming, often with structures designed to promote participation and introduce new contributors

Component coupling: Loosely coupled modules, frequently in a formal plugin system.

Main benefits: Builds a sustainable ecosystem in which the founding organization retains strong influence.

Typical governance: Benevolent dictatorship, with significant willingness to compromise to avoid forks.

Wide Open

EXAMPLES[9]

Rust (present day), *Apache HTTPD*

Characteristics: Wide Open projects actively welcome contributions from any source, and tend to get contributions at all levels: from individual developers providing one-off bug fixes to sustained organizational contributors developing major features and being involved in long-range project planning.

A Wide Open project values investing in contributors relatively highly: the core development team is generally willing to pay a high opportunity cost (in terms of code not written or bugs not fixed) in order to encourage or onboard a promising new contributor. The project's collaboration tools and processes reflect this prioritization. For example, the experienced developers in the project will still take the time to hold design discussions in forums-of-record — even when many of the individuals concerned are co-located and could make the decisions in person — in order to publicly reinforce the possibility of participation by any other self-nominated interested party.

Wide Open is a good archetype for projects that aim to serve the same technical purpose over a long period of time, and for which stability and reliability are more important than rapid development of the code base or fast adoption of field-specific innovations. Wide Open is also especially effective when some significant percentage of the user base is technically oriented (and thus susceptible to being converted to contributors), and when regular contact with a broad developer base is likely to bring real-world usage information that would be difficult for Mozilla to acquire in any other way.

Wide Open projects should expect the majority of contributors to be only lightly involved making the occasional small-scale contribution here and there, but not investing heavily in the project. A much smaller number of contributors will participate with higher degrees of intensity. Contributors tend to cycle in and out of these groups over time, and one of the strengths of the Wide Open archetype is that it makes it easy (from the contributors' point of view) for that cycling to happen, because the project as a whole is willing to sustain the overhead of maintaining permanent "on-ramps" to greater involvement.

In general, Wide Open projects usually settle on multi-party governance in some roughly democratic form. This is because when there are many contributors, some of them will expect to be able to climb the influence ladder by investing time and making quality contributions. Even for those who don't seek that climb, the knowledge that it's available – i.e., that past investment could, at some future point, be retroactively incorporated into a larger strategy of influence – is part of what makes working in a Wide Open project appealing.

[9] *In some ways the Linux kernel project could be considered "Wide Open". However, both technically and culturally, Linux kernel development is sui generis and we have deliberately avoided using it as the basis for any archetype.*

It is still possible for a single organization to retain significant and possibly decisive influence in a Wide Open project. The typical way to do so is to invest the most developer time. However, when an organization chooses that course, it must then be especially careful not to demotivate others by accidentally quashing their ability — real or perceived — to achieve influence. Simply declaring a project "Wide Open" and being receptive to contributions is not enough; ultimately, Wide Open projects work best when they are visibly influenceable by input from multiple sources, although one organization may be "first among equals" and remain able to exercise leadership in major decisions.

Licensing: Can be copyleft or non-copyleft.

Community standards: Very welcoming to contributors, to the point of having formalized systems for onboarding newcomers.

Component coupling: This archetype applies to many technically different projects, so the component organization could go any number of ways.

Main benefits: Enables large-scale collaboration. Community can become selfsustaining, which allows the original sponsoring organization some flexibility in deciding how involved to stay.

Typical governance: Group-based and more or less democratic. In practice, most decisions are made by consensus, but voting is available as a fallback when consensus cannot be reached. There is usually a formal committer group to approve contributions and anoint new committers.

Mass Market

EXAMPLES

Firefox, LibreOffice, MediaWiki
(due to Wikipedia instance)

Characteristics: Mass Market projects are often similar to "Wide Open" projects, but they necessarily have a protective layer around their contribution intake process. This is because there are simply too many users and too many opinions for everyone to be heard by the development team, and because the average technical sophistication of the users is low (especially in terms of their familiarity with the conventions of open source participation). This is especially true in a commercial context where product experience and time-to-market are highly sensitive.

Like B2B, Mass Market tends to drive down the price of directly competitive products, the way OpenOffice.org (and later LibreOffice) did with Microsoft Office. But Mass Market is less likely than B2B to affect complementary products, because Mass Market is usually aimed at individuals not at intermediary organizations.

Mass Market open source projects have certain opportunities that come with economies of scale, and they should actively seek to take advantage of those opportunities. For example, they can often get good translation (i.e., localization) coverage, because there are many users who are qualified and motivated to perform translation even though they might not have the skills to participate as programmers. Mass Market projects can do effective A/B testing and user surveys, and have some ability to influence ***de facto*** or formal Internet and Web standards. Finally, they can be thought leaders in a broader political sense (e.g., through industry partnerships, user clubs, etc).

Mass Market might at first seem more like a description of an end product than of a project archetype, but in fact there are important ways in which being Mass Market has implications for how the project is run. With developer-oriented software, it usually pays to listen to the most vocal customers. But Mass Market projects are oriented mainly toward nontechnical users, and thus should be guided more by explicit user research.

Licensing: Non-copyleft generally, but may be copyleft depending on the project's business strategy.

Community standards: Fully open, but relatively brusque for the vast majority of users. Because of the large number of users, these projects evolve toward a users-helping-users pattern, rather than the development team serving as user support. The team itself can often seem somewhat distant or unresponsive.

Component coupling: Mix and match packages. This archetype can support a number of different component packaging techniques.

Main benefits: Large user base can help the project be broadly influential.

Typical governance: Technical committee and/or formal committer group.

Specialty Library

EXAMPLES

libssl, libmp4

Characteristics: Specialty libraries provide base functionality to a set of functions whose implementation requires specialized knowledge at the leading edge of research. For example, developers shouldn't roll their own crypto or codecs; instead, everyone benefits from easy access to a pooled set of code built by experts.

These libraries are fairly commoditized, as the set of capabilities for such a library is fairly fixed. Any ssl or mp4 library is going to do roughly the same thing. Competing libraries might differ at the API level, but there is no arms race over features. Given the lack of opportunity for competitive advantage in developing one's own library, most of those capable of contributing have every reason to do it as part of an open source project that shares pooled development costs and benefits.

Widespread adoption of a specific open source library can power standardization work and also increase adoption of formal and informal standards. Coupled with careful approaches to patents, this can open up access to participants who aren't attached to companies with deep pockets and large patent portfolios.

Specialty libraries look like open source projects with high barriers to entry. Anybody can submit a patch, but landing it might require a high degree of expert work before it is accepted into core. Coding standards are high. There is often less developer outreach and hand-holding of new developers as participants are largely assumed to be experienced and capable. In a heavily patented area or one subject to export controls, acceptance policies might be more complex and even involve legal review.

Licensing: Usually non-copyleft.

Community standards: High barriers to entry, largely because contributors are expected to be experts.

Component coupling: Tightly coupled. These libraries are structured to do one thing well.

Main benefits: Ensures a shared solution to a specific technical problem. Cooperation at the engineering level can lead to new organizational partnerships.

Typical governance: Formal committer group that can grant committing privileges to new contributors.

Trusted Vendor

EXAMPLES

MongoDB, Hypothes.is, Coral

Characteristics: Overall project direction steered by a central party — usually the founding organization — but with encouragement of an active commercial ecosystem based on the project, thus requiring occasional restraint or non-competitive behavior on the part of the primary maintainer.

The guiding principle behind the Trusted Vendor archetype is that nobody can build everything. Sometimes people just want a complete solution from a vendor. Switching costs for such products are often high, so choosing a proprietary solution could lock one into a technology and a vendor. That lock-in and the related dependence gives vendors power over their customers and users, which allows vendors to overprice and underdeliver. Customers and users too often find themselves trapped in a relationship that doesn't meet their needs and sometimes feels abusive.

Open source is the antidote to vendor lock-in. First, it enables competition. If a vendor fails or pivots or suddenly raises prices, a functioning open source ecosystem can provide a replacement without high switching costs. Second, open source ecosystems tend toward open standards. Integration, custom development, or even switching all come easier, faster and cheaper if the underlying solution makes good use of common standards.

This connection between open source and risk-free vendor dependence is strong enough to shift customer preferences. More and more procurement processes are aimed at open source solutions precisely because they avoid lock-in during long-term vendor relations.

There are other elements of trust besides just defusing the threat of lock-in. Some of the benefits described for the "Wide Open" archetype are also present in Trusted Vendor. In the latter case, though, those benefits are often mainly signaling mechanisms: even if customers never take direct advantage of the available collaboration mechanisms, they are reassured by the existence of those mechanisms.

However, as products mature, fulfilling the promise of those mechanisms can become a prerequisite for jump-starting the third-party vendor and integrator ecosystem, which may eventually cause a project to shift to something more like the Multi-Vendor Infrastructure archetype. In Trusted Vendor projects, the primary maintainer retains more control over project direction than in Multi-Vendor Infrastructure. The commitment to being an honest broker requires the maintainer to make certain commitments about what it will and won't do with that control, both in terms of the project's technical roadmap and in terms of allowing — indeed, encouraging — other vendors to flourish by basing their own services on the software. A Trusted Vendor must actively avoid exclusive or monopoly powers and even avoid some of the competitive privileges that usually come with being a project's prime mover.

For Mozilla, this archetype can be an attractive differentiating advantage when building services or infrastructure for the open Web. A deep commitment to privacy, data access, and open standards all mesh well with the need to build trust that allows low risk dependence.

Licensing: Can be copyleft or non-copyleft; copyleft is often used to discourage competition from forking.

Community standards: Users and contributors have input to roadmap, though they may not contribute directly.

Component coupling: Tightly coupled.

Main benefits: User community — including deployers and commercial support vendors — tends to be long-term, which provides both stability and word-of-mouth marketing to the project.

Typical governance: Maintainer-led by the vendor.

Upstream Dependency

EXAMPLES

OpenSSL, WebKit

and just about every small JavaScript library on GitHub

Characteristics: Intended as a building block for other software, so the user base is made up of developers, and adoption means third-party integration of your software into their products.

For smaller or more specialized libraries, this is a bit like Wide Open, but the experience for maintainers and contributors is qualitatively different. When a significant portion of the user base is able to participate technically, a lot of attention and investment may be needed from each serious participant just to take part in discussions, even when the number of parties in a discussion is small. Relative to Wide Open, there are fewer one-off or lightweight contributors and more ongoing, engaged contributors.

The degree of the software's specialization does not necessarily dictate how much investment it receives. Upstream Dependency projects attract technical investment based on two factors: their replaceability (it's easier to duplicate or find an alternative for a six line JavaScript library than it is to do so for, say, all of OpenSSL), and which downstream projects depend on them.

Therefore understanding who is using the project, and how, is key to knowing what goals the project can serve for Mozilla. An Upstream Dependency project can be quite influential thanks to its role in downstream dependees, even if only a small minority of developers in those dependees are familiar with it.

Licensing: Typically non-copyleft.

Community standards: Welcoming, and specifically amenable to one-time contributions.

Component coupling: Standalone, decoupled from one another and the downstream projects that pull them in.

Main benefits: Connections to many downstream dependee projects offers insight into market and usage trends, and can provide openings to potential partners.

Typical governance: Informal, maintainer-led, committer groups.

Bathwater

Characteristics: This is code dumped over the wall. It is purely about distribution, not about any particular mode of production. Think of it as the ground state of open source projects: someone publishes code under a free license but invests no followup effort into building open source dynamics.

Typically, the published code is not the organization's prized technology. In some cases it is a one-off publication and thus likely to become quickly outdated. In other cases, there may be a public repository, but that repository is not where development occurs — instead it is just a container for periodic snapshots of particular release points. Sometimes there is not even a public repository, and the only thing published is a release tarball or a sequence of them.

Code gets dumped this way quite often[10], and "throwing code over the wall" has a bad reputation among open source practitioners. Merely publishing code under an open source license is unlikely to generate any of the beneficial effects that come from truly investing in open source. When those benefits don't materialize, developers sometimes take it as evidence that open source "doesn't work".

Although experienced open source professionals look down on Bathwater projects, they do have some uses. First, they work as an initial foray into open source. For inexperienced firms testing the waters, there are few benefits from open sourcing this way but also few risks. They can claim to be doing open source, send signals that they are open to more open source engagement, and perhaps that positions them to do a more complete job the next time around.

Second, even for firms experienced in open source, Bathwater is a reasonable final state for an abandoned initiative. When an organization is ready to stop devoting any resources to a project, explicitly putting the code into Bathwater mode, and giving the world permission to take it up, can be a final attempt to give the project a chance at impact and serve the existing user base.

Bathwater treats the code like a forkable resource. Users of Bathwater projects don't expect anything from upstream, because there is no upstream anymore, but they can can consider their involvement as a greenfield opportunity. If they are willing to invest in the code, they can choose any one of the other archetypes and start building toward it.

[10] *Although numerous, Bathwater code bases tend not to be famous, because they don't get much usage unless they attract an open source maintenance community – at which point they transform into one of the other archetypes. One recognizable example of Bathwater code is https://github.com/Conservatory/healthcare.gov-2013-10-01 which was essentially a one-time dump of the front end web code for the original http://healthcare.gov site developed by the U.S. government. (Although technically in the public domain, that code could in principle have been adopted by anyone and the resulting fork placed under an open source license. As is often the case with Bathwater code, that uptake did not happen.*

Licensing: Varies, sometimes missing, often incoherent.

Community standards: Community is dormant to non-existent.

Component coupling: Depends on the project, but in practice often standalone spaghetti.

Main benefits: Doesn't cost anything to run.

Typical governance: None.

Evolution From One Archetype To Another

Projects can change from one archetype to another, and often do. It may be a deliberate and guided transformation, or it may be a gradual and incremental evolution that is not driven by any one participant.

An example of a deliberate transition is Firefox, which went from Rocket Ship To Mars (or something close to it) to a mode much closer today to Mass Market. An example of a less centrally-driven and more incremental transition is the WordPress open source project, which has been moving gradually from something like Trusted Vendor to something more like Controlled Ecosystem over a long period of time. WordPress's transition cannot be said to be purely natural, as it has occurred with the acquiescence and probably the encouragement of WordPress's primary commercial backer, Automattic, Inc — and this is probably the common case: with the exception of genuine hard forks, it is rare for an archetype transition to occur entirely against the wishes of the most invested community members, but that does not mean that every such transition is necessarily chosen or driven by those members either.

In general, transformations from Rocket Ship To Mars to some other archetype are among the most common, because they reflect a natural progression: from a small, focused, and unified core team working on something new, to a position where more people are involved and have broadening influence in the project. If those new arrivals are mostly developers whose motives are immediate rather than strategic, that transformation might be to ward Specialty Library. If they are developers using the code as part of other programs, it might be toward Trusted Vendor or Upstream Dependency. If they are plugin developers and API consumers, it might be toward Controlled Ecosystem. If they are organizations or organizationally-motivated developers, B2B or Multi-Vendor Infrastructure might be likely places to end up. If they are users, or a combination of users and developers, that might point the way toward Mass Market or Wide Open.

Rocket Ship To Mars is not the only starting point for archetype transitions. A project might move from B2B to Upstream Dependency as it becomes commoditized, for example, perhaps through competition or perhaps just through rapid adoption. We will not attempt here to map out the full combinatoric range of possible archetype transitions. Having such a map would not be very useful anyway. What is useful instead is to be always alert to the possibility of transformation, and to ask, at strategic points in a project's life cycle, whether a purposeful transformation might be called for, and whether there might be bottom-up pressure pushing the project toward a particular new archetype.

Quick-Reference Comparison Of All Archetypes

	B2B	Multi-Vendor Infra	Rocket Ship to Mars	Controlled Ecosystem	Wide Open	Mass Market	Specialty Library	Trusted Vendor	Upstream Dependency
Main benefit	Driving industry adoption of your technology	Collaboration with partners; address a set of shared problems	Quick, focused effect in a specific area	Can build a sustainable ecosystem in which founding organization has strong influence	Large-scale collaboration; community can become self-sustaining	Large user base can make project broadly influential	Ensure quality solution to a specific problem; can lead to new partners	Loyalty of downstream consumers helps project stability	Broad reach across (hence insight into) many dependee projects
Main drawback	Little or no collaborative development	Sometimes off-putting to individual contributors	Collaboration only available from those who share a very specific vision	Compromise needed to avoid forks (esp commercial)	Effort to maintain onboarding paths & manage all participants	Huge user base needs filtering for dev community	High barriers to entry; relatively small developer pool	Primary org must be careful how it uses its position	Developer base can sometimes be lightly motivated
Main situational consideration	Requires major market power to be effective	Business needs of participants affect community management	Everything depends on success of original vision	Participants have many motivations (commercial & non-commercial)	Differing commitment & engagement levels among participants	Contributor base does not accurately represent user base	Standard-setting effects (de facto or official)	Customer needs vs open source project needs	Usage patterns of downstream consumers
Development speed	Fast; pace set by business goals	Usually moderate, but depends on needs of participants	Fast; escape velocity	Medium	Slow medium; some process overhead	Slow medium; swift change destabilizes user base	Gets slower over time, as library stabilizes	Medium: primary vendor momentum vs third-party needs	Medium; may slow down as standard settles
Component coupling	Tightly coupled, to market one unified product	Loosely-coupled with de facto standards	Tight, to ship one core product	Loosely coupled, often with plugin system	Variable	Variable	Tightly coupled	Tightly coupled	Standalone; decoupled from one another & from downstream projects
Typical participant type	Organizational reps	Organizational reps	Founding organization	Founder, some external core contributors, many plugin contributors	Open to anyone; participant demographic depends on project	Organizational reps, redistributor reps; some users who are technical	Developers with expertise in the relevant field	Customer reps (both paying and non-paying); some one-off contributors	Downstream devs
Typical licensing	Almost always non-copyleft	Usually non-copyleft, because many internal forks	Usually non-copyleft, but with occasional exceptions	Any, but requires thought re plugins	Any	Usually non-copyleft, but depends on business strategy	Usually non-copyleft	Either; copyleft sometimes used to discourage competitive forks	Usually non-copyleft
Ease of onboarding	Hard	Medium	Hard	Medium	Easy	Easy to medium	Depends on technical complexity	Medium to hard	Depends on technical complexity
Community standards	Oriented toward organizations	Welcoming but formal; difficult for individuals	Focused on core group	Welcoming, with some onboarding structures	Very welcoming, formalized onboarding systems	Fully open; scales via users helping users	High barrier; contributors need to be experts	Clear boundaries: users have mainly roadmap input	Welcoming; amenable to one-time contributions
Typical governance	Lead company plus partners	Committee of organizational reps	Founder governs with iron fist	Benevolent dictatorship; tries to avoid forks	Group-based; consensus / democratic	Main organization leads, with outside input	Multi-party committer group	Main vendor leads	Informal, maintainer-led, committer groups
Measures of open source success	1. Adoption by target partners (aiming for dominant levels) 2. Code contributions from partner organizations 3. Successful projects built around core project	1. Partners contributing at equal and high levels 2. Longevity of contribution by committers 3. Variety of organizations contributing	1. Speed of development 2. Adoption by target users 3. Achieving original technical goals	1. Adoption by target users 2. Increase in number of extension developers 3. Overall cohesiveness of community	1. Growth in number of contributors 2. Efficiency of contribution 3. Variety in where decisions get made	1. Awareness among users that project is open source 2. Growth in non-technical contributors 3. Effective filtering of user feedback to developers	1. Adoption by intended domain (perhaps even dominant adoption) 2. High quality of contributors 3. High quality of code	1. Absence of competitive forks 2. Business needs of vendor served 3. Vendor's leadership accepted by community	1. Multiple competitive distributions 2. Longevity of participation 3. Bug reports tend to be technical and constructive
Examples	Android; Chromium	Kubernetes; Open Stack	Meteor; Signal	WordPress; Drupal	Rust (present day); Apache HTTPD	Firefox; MediaWiki (due to Wikipedia instance)	libssl; libmp4	MongoDB; Hypothes is; Coral	OpenSSL; WebKit; just about every small JavaScript library on GitHub

Practical Questions To Ask About A Project

This section presents a checklist of practical questions that can either confirm a choice of archetype or, perhaps, help draw out latent disagreements about what archetype is best for a given project at a particular stage in its development.

We emphasize again the baseline definition:

> *A software project is open source if its code is distributed under an OSI and FSF-approved license to parties who could use and redistribute the code under that license.*

That's all that is absolutely necessary for open source. Everything else is a matter of choice and thus a matter of potential variation across archetypes.

For example, a core development team is not required to review or accept code contributions from outside developers. The core team might ***choose*** to engage with outside contributions because it is advantageous to do so, but that is independent of the question of whether the project is open source. Similarly, it is not necessary to give users or contributors any voice in project governance — that is, governance of the particular master copy from which your organization cuts releases — but it might under some circumstances be a good idea to give them that voice.

Questions to Consider

How many separable open source pieces are there?

For many organizations, there is a strategic question of how much to open source for each product. This question is easier at Mozilla than it is at most other places, since Mozilla has one of the most aggressive policies of open sourcing its software in the industry[11]. But there is still the possibility that different parts of what looked initially like one project might be better run as separate projects, perhaps using different archetypes.

For example, one could imagine a project where the client-side code is developed with heavy community involvement and a multi-organizational governance process, while the server-side (perhaps primarily deployed as a single centralized service by Mozilla or its partners) is run in a much more dictatorial way, with outside contribution limited mainly to requests for APIs. Even among modules within the same high level component, it can sometimes make sense to use different archetypes: core cryptographic code, for example, may need different governance and receptivity to contribution than user interface code does.

[11] https://www.mozilla.org/en-US/foundation/licensing/

Practical Questions To Ask (continued)

Additionally, there may be questions about how to release the non-code parts of a project. Documentation, the project website, and media assets can all be released under open source licenses[12]. Trademarks too can be approached in ways that are compatible with certain kinds of use by the broader open source community, including commercial use. These assets merit separate consideration even when the code itself is to be run as one project.

Who is expected to participate in the project, and why?

In this context, "participate" means to contribute effort directly to the project in a way that results in tangible improvements to the final, shipped product.

- Individually motivated developers?

  Is the project optimizing for low-investment ("drive-by" or "one-off") contributors, or more for high-investment developers who will have a sustained presence? How broad or narrow is the product's audience? Do developers need unusual skills and knowledge to participate?

- What about documenters? Testers? Translators? UX experts? Security reviewers? Domain experts specific to the software's areas of applicability?

- Organizational participation?

  While organizations often participate indirectly via their developers, some also participate at a higher organizational level with more explicit executive buy-in. Projects that intend to have organizational participation often use an archetype that is compatible with governance structures designed for organizational participation.

- Is the user base expected to be the primary pool for the contributor base?

- Who will use this? Is it aimed at a commercial sector? Is it infrastructure? What else already exists in that sector, and what are their sustainability approaches?

How is the code expected to be deployed?

The way(s) in which a product is deployed can affect what modes of open source production will be best for that project. For example, optimizing for broad developer participation in a project that is expected to have essentially one deployed instance would usually not make sense, except in special circumstances.

[12] *We may consider the Creative Commons Attribution, Attribution-ShareAlike, and CC0 licenses to be open source licenses for these kinds of assets.*

Practical Questions To Ask (continued)

- Client-side deployment for each user?
- Web application? Back end application with rich APIs?
- Broad cloud deployment on the public Internet?
- Enterprise-driven behind-the-firewall (BTF) deployment?
- A small number of relatively high-investment cloud instances?
- Just one main cloud instance (single-entity SaaS or PaaS)?

How is the project managed and governed?

- Does the founding organization (which may or may not be Mozilla) want to maintain tight control?
- Where does Mozilla want this project to be on the tradeoff between focused roadmap and iterative consensus-building?
- Informal versus formal organizational participation?
- What level of risk of third-party forks is acceptable?
- Is the project primarily developer-driven, or is it driven by a product vision extrinsic to the core developers?

What is the project's business model or sustainability model?

Not all projects necessarily have a sustainability model at first, but it is worth asking this question and thinking about how it might affect, or be affected by, choice of archetype. For example, a project whose long-term future depends on deep investment from a few key industry partners might do better to aim for B2B or Multi-Vendor Infrastructure than for (say) Mass Market.

Which open source license does it use?

See "Basics of Open Source Licensing" for more on this topic.

Practical Questions To Ask (continued)

- Main question: how much copyleft is appropriate? None, some, or as much as possible?
    - For some situations: "weak copyleft" (e.g., the Mozilla Public License (MPL), or the LGPL). So-called "weak copyleft" is used by software libraries to preserve copyleft for the original library, while not enforcing copyleft on co-distributed code that uses the library but is not actually part of the library.
    - For some situations: on-contact copyleft (e.g., AGPL). In this choice, copyleft provisions travel with distribution, and "distribution" is defined by the license to include even making use of the software via a network connection.
- Does the project involve any trademarks? If so, a separate trademark policy may be necessary in addition to the open source license. Trademarks operate quite differently in open source ecosystems, and the considerations in this area can be especially tricky.
- Is the project operating in a heavily patented area? If so, a license with patent provisions may be advisable (see the discussion of "patent snapback" in "Elements that vary across open source licenses").
- What level of formality will be used for accepting incoming contributions? (E.g., CA, CLA, or DCO. See the discussion of contributor agreements in "License Enforcement and Receiving Contributions".)

Does technical architecture match social structure?

- Technical architecture can have effects on governance, licensing, and overall project/community ecosystem.
    - Is the project module-based with APIs, or is it a microservice architecture?
    - Does the project involve a run-time extension language?
    - Will the project encourage a semi-independent plugin ecosystem?
    - Do the project's choice of programming language(s) and dependencies match the intended archetype?
    - Are the project's collaboration tools and processes configured to match the intended archetype?

Basics of Open Source Licensing

Although many Mozillians have a great deal of familiarity with the intricacies of different open source licenses, not everyone does. This section provides the basic vocabulary and concepts needed to understand the most important choices in open source licensing; it is not a complete guide to open source licensing, however, and it is not legal advice.[13] You should consult your organization's legal department on all licensing and other legal matters.

Elements Common To All Open Source Licenses

All open source licenses offer a core set of fundamental and non-revocable rights, granted by the licensor (the copyright owner of the code) to the recipient (the "licensee"):

- Anyone can ***use*** the software for any purpose.
- Anyone can ***modify*** the software's source code. (This means, by the way, that source code is the thing to which an open source license applies. It would make no sense to release binary-only software under an open source license, and if one were to do so, no one else would consider it to be an open source release.)
- Anyone can ***redistribute*** the software, in both original and modified form, to anyone else, under the same license.

Virtually all open source licenses also include a prominent disclaimer of liability, so that the licensor cannot be held responsible if the code (even in unmodified form) causes a recipient's computer to melt.

Elements That Vary Across Open Source Licenses

In addition to the core freedoms described above, some open source licenses include some additional conditions. Usually these conditions place certain demands on the recipient, or constraints on the recipients' behavior, in exchange for the recipient's right to use and redistribute the code.

copyleft vs non-copyleft (a.k.a. reciprocal vs non-reciprocal)

Some open source licenses have a **copyleft** (also called **reciprocal**) license provision, meaning that derivative works (i.e. works based on the original) may only be distributed under the same open source license as the original. Furthermore, copyleft says that under certain circumstances distribution must take place: that is, your right to use the

[13] *Some of the material here is adapted from https://producingoss.com/en/legal.html, which is also not a complete guide to open source licensing, and is likewise not legal advice, but does offer more details and pointers to further resources.*

Basics of Open Source Licensing (continued)

code depends on the license you received it under, and in a copyleft license that right is contingent on you redistributing the code to others who meet certain requirements and request redistribution.

In practice, this boils down to the question of whether the code can be used in proprietary products. With a **non-copyleft** or **non-reciprocal** license[14], the open source code can be used as part of a proprietary product: the program as a whole continues to be distributed under its proprietary license, while containing some code from a non-proprietary source. The Apache License, X Consortium License, BSD-style license, and the MIT-style license are all examples of non-copyleft, proprietary-compatible licenses.

A copyleft license, on the other hand, means that when the covered code is included in a product, that product can only be distributed under the same copyleft license — in effect, the product cannot be proprietary.

There are some complexities to this, which we will only touch on here. One widely used copyleft license, the GNU General Public License (GPL), says more or less that distribution of source code under the GPL is required only when the corresponding binary executable is distributed. This means that a SaaS product can include GPL'd code, and as long as the actual executable is never distributed to third parties, there is no requirement to release the entire work under the GPL. (Many large SaaS companies are actually in this position.) A subsequent variant of the GPL, the Affero GPL (AGPL), closes that gap: a user who merely accesses the program's functionality over a network thereby acquires the right to request full source code under the AGPL.

Mozilla's own license, the Mozilla Public License, is a kind of half-copyleft (sometimes called "weak copyleft") open source license. It requires redistribution of derived works under the MPL, but has a very narrow definition of what constitutes a derived work: essentially, modifications to the original files distributed under the MPL must be released under the MPL, but surrounding code that merely calls code from those files may remain under a proprietary or other non-MPL license.

When thinking about copyleft vs non-copyleft, it is helpful to keep in mind that "proprietary" is not the same as "commercial". A copyleft program can be used for commercial purposes, just as any open source program can be, and the copyleft license permits charging money for the act of redistribution as long as the redistribution is done under the same license. Of course, this means that the recipient could then start handing out copies for free, which in

[14] *Sometimes people call non-copyleft licenses "permissive" licenses. We have avoided that terminology here because it seems to imply that copyleft licenses are somehow non-permissive, which would be inaccurate. Copyleft licenses grant all the same rights as non-copyleft licenses, and merely ask that one refrain from imposing restrictions — that is, refrain from behaving non-permissively — toward others downstream.*

Basics of Open Source Licensing (continued)

turn makes it difficult for anyone else to charge a lot of money for a copy, so in practice the price of acquiring copies is driven to zero pretty quickly.[15]

attribution

Most open source licenses stipulate that any use of the covered code be accompanied by a notice, whose placement and display is usually specified, giving credit to the authors or copyright holders of the code.

These attribution requirements tend not to be onerous. They usually specify that credit be preserved in the source code files, and sometimes specify that if a program displays a credits list in some usual way, the open source attribution should be included in those credits.

protection of trademark

(This may be thought of as a variant of attribution requirements.)

Trademark-protecting open source licenses specify that the name of the original software — or its copyright holders, or their institution, etc. — may not be used to identify derivative works, at least not without prior written permission.

This can usually be done entirely via trademark law anyway, without reference to the software's copyright license, so such clauses can be somewhat legally redundant. In effect, they amplify a trademark infringement into a copyright infringement as well, thus giving the licensor more options for enforcement.

patent snapback

Certain licenses (e.g., the GNU General Public License version 3, the Apache License version 2, the Mozilla Public License 2.0, and a few others) contain language designed to prevent people from using patent law to take away open source rights that were granted under copyright law.

These clauses require contributors to grant patent licenses along with their contribution, usually covering any patents licenseable by the contributor that could be infringed by their contribution (or by the incorporation of their contribution into the work as a whole). Then the open source license takes it one step further: if some one using software under the open source license initiates patent litigation against another party, claiming that the covered work infringes, the initiator automatically loses all the patent grants from others provided for that

[15] *But see Red Hat, which distributes open source software mingled with proprietary elements that encumber redistributing entire copies of RHEL.*

Basics of Open Source Licensing *(continued)*

work via the license, and in the case of the GPL-3.0 loses their right to distribute under the license altogether.

tivoization

"Tivoization" (or "tivo-ization") refers to the practice of using hardware-enforced digital signatures to prevent modified open source code from running on a device, even when that device natively runs an unmodified version of the code — a version that, because it is unmodified, can pass the signature check.[16]

Most open source licenses permit tivoization, including most of the copyleft licenses and in particular the GPL version 2. However, the GPL version 3 and the AGPL have some limited anti-tivoization measures applicable to consumer and household products. The effect of these measures on Mozilla software is far beyond the scope of this document. If tivoization might be relevant to your open source project, we recommend obtaining legal advice as early as possible.

License Compatibility

The non-copyleft open source licenses are generally compatible with each other, meaning that code under one license can be combined with code under another, and the result distributed under either license without violating the terms of the other.

However, the copyleft vs non-copyleft distinction can lead to **license compatibility** issues between a copyleft license and other licenses — sometimes even with respect to a different copyleft license.

This report does not include a full discussion of license compatibility issues, but such issues should be taken into account when starting any new project. One of the most important questions to ask about a new project is what other licenses the project's code will need to be able to be combined with.

[16] The name comes from TiVo's digital video recorders, which runs the GPLv2-licensed Linux kernel and uses this technique to prevent people from running versions of the kernel other than those approved by TiVo. See https://en.wikipedia.org/wiki/Tivoization for more.

Basics of Open Source Licensing (continued)

License Enforcement and Receiving Contributions

For all open source software — indeed, for any copyrighted work — the copyright holder is the only entity with standing to enforce the license.

For example, in an archetype like Rocket Ship To Mars or Controlled Ecosystem, the founder or primary vendor is often the only entity holding copyright in the code. If they release that code under a copyleft license, they can ensure that anyone who forks the project or distributes a derivative work built on it abides by the license.

The complication, however, is that once a project starts accepting code and documentation contributions from others, copyright ownership becomes spread out among all those authors unless they sign an agreement to transfer copyright to a central owner (which is, for various reasons, less and less common a practice). The exact situation for a given project will depend on whether it requests a formal contributor agreement from its participants and, if so, what kind. More and more projects are using a very simple kind of contributor agreement known as a “Developer Certificate of Origin” (DCO).

This report does not include a in-depth discussion of contributor agreements and copyright ownership issues, but https://producingoss.com/en/contributor-agreements.html has more information and pointers to further resources.

Note that non-owning parties cannot enforce the license on an owner. In other words, if Mozilla releases a project under the AGPL, Mozilla itself is not obliged to abide by the terms of the AGPL at least as long as Mozilla is the sole copyright holder (though in practice Mozilla should conspicuously abide by those terms anyway, of course, for strategic reasons that go beyond its formal obligations as licensor).

Methodology

To prepare this report, we interviewed about a dozen Mozilla-affiliated people with diverse views on open source across a range of Mozilla initiatives and products. Initially, those interviews focused on a few Mozilla efforts (Firefox, Rust, and Servo), but eventually they grew to include discussions of many Mozilla efforts and delved into policy beyond just open source. We also drew on examinations of open source projects unaffiliated with Mozilla; we initiated some of those examinations specifically for this report, and had already done others as part of previous research efforts.

Acknowledgements

We are grateful to all of the interviewees at Mozilla who gave generously of their time and expertise. We have not named them here, in part because some asked not to be named (in order to speak frankly), and in part because each archetype should stand on its own and not be associated with any particular person.

Work On ’Openness’ At Mozilla

In 2017, Mozilla’s Open Innovation team completed a thorough review of the use of “openness” as a competitive advantage in Mozilla software and technology work (“Open by Design”), and the need for a report such as this was one of the findings. More information about this previous review can be found at https://medium.com/mozilla-open-innovation.