moz://a

Open Tech Strategies

OPEN SOURCE ARCHETYPES:

A Framework For Purposeful Open Source

Version 2.0 *November 2019*

Open Source Archetypes: A Framework For Purposeful Open Source

https://opentechstrategies.com/archetypes

Version 2.0

28 October 2019

Preface **3**

Changes from Version 1 to Version 2 **4**

Introduction **5**

How to Use This Document **6**

Setting Open Source Goals **7**

Open Source Project Archetypes **9**

Archetypes At A Glance **26**

Contributors and Archetypes: Setting Expectations **27**

Signs of Health: Metrics and Archetypes **28**

Ecosystem Mapping **31**

Business Models and Archetypes **35**

Transitions: Changing From One Archetype To Another **37**

Choosing An Archetype: Practical Questions **38**

Methodology **41**

Acknowledgements **41**

Appendix A: Basics of Open Source Licensing **42**

Appendix B: Work On "Openness" At Mozilla **46**

Copyright **46**

Preface

Version 1 of this report was originally commissioned by the Mozilla Corporation for internal purposes. Mozilla wanted a shared framework for discussing how to choose its open source investments projects. 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.

Realizing the benefit of having a common, public vocabulary of archetypes, Mozilla decided to release the report publicly. Very little was changed between the internal version 1 and the public version 1. We left the Mozilla-specific orientation intact, on the theory that the analysis would be clearer if tuned to a specific (and well-known) organization rather than rewritten for a hypothetical general audience.

The first edition stimulated a fair amount of discussion, both inside and outside Mozilla. In particular, in the course of applying archetypes-based analysis to various open source projects, Mozilla found places where the report could go into more depth or cover certain adjacent topics. Mozilla's thus commissioned an updated Version 2. The major differences are listed in the section "Changes from Version 1 to Version 2" on p. 4.

As with Version 1, we believe that this is and should always be a work in progress. There are probably archetypes we have not thought of, and for the archetypes we have identified there may be important properties not discussed here. When we published version 1 we wanted it to be part of a larger conversation about open source project archetypes, and fortunately that's what happened. We intend the same for Version 2, with the added desire that that conversation will grow to include discussion of how projects can use archetype analysis to guide their open source investments and overall strategy.

Changes from Version 1 to Version 2

Those familiar with version 1 of the report may use this table to find the major new material in version 2. Although most sections were touched in the update, the changes to existing sections — especially the archetypes themselves — were generally minor.[1] Instead, we focused on providing more information about how to *use* the archetypes. That material is in the new sections listed below, so they would be a natural place to start for those who have already read the first version of this report.

	"Preface"
	"Introduction"
	"How to Use This Document"
(removed)	"Benefits of Open Source" *(replaced by new Goals section below)*
(new section)	**"Setting Open Source Goals"**
	"Open Source Project Archetypes"
	- "Business-to-Business (B2B) Open Source"
	- "Multi-Vendor Infrastructure"
	- "Rocket Ship To Mars"
(new archetype)	- **"Single-Maintainer Houseplant"**
	- "Controlled Ecosystem"
	- "Wide Open"
	- "Mass Market"
	- "Specialty Library"
	- "Trusted Vendor"
	- "Upstream Dependency"
	- "Bathwater"
(heavily revised)	**"Archetypes At A Glance"** *(formerly "Quick-Reference Chart")*
(new section)	**"Contributors and Archetypes: Setting Expectations"**
(new section)	**"Signs of Health: Metrics and Archetypes"**
(new section)	**"Ecosystem Mapping"**
(new section)	**"Business Models and Archetypes"**
(new section)	**"Transitions: Changing From One Archetype To Another"**
	"Choosing An Archetype: Practical Questions"
	"Appendix A: Basics of Open Source Licensing"
	"Appendix B: Work On "Openness" At Mozilla"

[1] The exception is "Archetypes At A Glance" on p. 26, which was called "Quick-Reference Chart" in version 1. It has been renamed and extensively revised for version 2.

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[2] and an open source license according to the OSI[3][4]. 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 nor 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.

---

[2] *https://fsf.org*

[3] *https://opensource.org*

[4] 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" (p. 38), 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 (listed in section "Open Source Project Archetypes", p. 9) 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 purpose 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 objectives 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 "Setting Open Source Goals" (p. 7). Which ones, if any, resonate with your project's purpose and needs? Note that no project should seek (or be expected to obtain) all of the possible benefits. The point of the review is to stimulate thought about which benefits are most important for this particular project — and for which you should therefore optimize.

3. Based on an understanding of project's purpose 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 "Choosing An Archetype: Practical Questions" (p. 38) to help determine if the archetype seems suitable, or 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[5], though in general we expect most projects will fit best with one archetype. Once that archetype is identified, governance, licensing, process, and infrastructure decisions can be made in light of it, with some adjustments for the project's specific circumstances.

[5]For example, Open Stack is listed as an example for both the "Wide Open" and "Multi-Vendor Infrastructure" archetypes because it exhibits characteristics of both: significant project-wide investment in welcoming individual contributors and giving them influence, but still with the majority of overall development activity funded by organizations that have a sustained business interest in the project's success.

Once you are familiar with the archetypes, the questions in "Choosing An Archetype: Practical Questions" may be useful to consider, along with the analysis exercises in "Setting Open Source Goals" (p. 7) and "Ecosystem Mapping" (p. 31).

Setting Open Source Goals

One of the keys to using open source successfully is setting appropriate goals. While open source is not a panacea, there are certain circumstances and problems to which it is very well suited. This section contains a list of goals that we have seen open source address successfully. We intend it to be all-encompassing for typical use — that is, this list should cover the goals that an open source project can reasonably achieve in the vast majority of circumstances. If you have a goal that isn't captured by this list, it might be a poor match for open source as typically practiced.[6]

Development and Collaboration

- Expand or amplify developer base, especially non-staff developers
- Gain market insight
- Gain or maintain insight in a particular technical domain
- Influence a technical domain
- Create a framework for partner onboarding and collaboration
- Lead a standardization effort
- Disrupt an incumbent, hold off insurgents

External Positioning

- Ease customer fears of vendor lock-in
- Deepen engagement with users, create more paths for engagement
- Increase transparency for customers and partners
- Establish a basis for product reputation
- Support organizational branding and credibility

[6]Is it also possible, of course, that the list is incomplete. If you think we missed something, please let us know at *https://www.OpenTechStrategies.com/archetypes/feedback*.

Internal or Structural Change

- Improve internal collaboration (cross-individual or cross-departmental)
- Create opportunities for internal mobility
- Expand or reshape developer hiring pool
- Improve morale and retention
- Create paths of flow for bottom-up innovation
- Improve and maintain open source capabilities (both technical and social)

Before you start picking goals off this list, take a moment to reflect on the overall purpose of your effort. Open source is one toolset in an overall strategy, and these open source goals only have value in service to a larger vision. Know your mission, your cooperative and competitive landscape, your short- and long-term milestones, and the resources you have at your command. That context should tell you a lot about what you need from your open source investment, and it should help narrow your goals to an achievable list.

All eighteen goals are probably plausible, to some degree, for any given open source project. Your project cannot, however, move in eighteen directions at once.

Think about which of the goals would fill important gaps in your larger strategy. Pick a few goals based on that — we normally say three, and although that's not a hard-and-fast limit you should think carefully before exceeding it.

Then match those selected goals up with archetypes that suit them. If the goals you've chosen align with archetypes that fit your project and fit your overall mission, that's a good sign you've found a path that has served other projects successfully. If you cannot find that alignment, look harder at some of the other possible goals, or consider different archetypes or blended archetypes.

Finally, document your goals: make them explicit and discuss them with your team. The more clarity people have about how the strategy drives the work, the better they can fine tune the work to your real goals. Consider what indicators you can use to establish starting points and targets. We cover metrics elsewhere (see "Signs of Health: Metrics and Archetypes", p. 28), but you don't need quantifiable indicators at this stage anyway. You just need to be able to answer the questions "How will we know if it's working?" and "What signs might indicate that we need to make adjustments?"

Open Source Project Archetypes

The archetypes described in this section are simplified versions of those found in the wild.[7] 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 to ever use itself.

These archetypes are primarily a vocabulary that enables us to label the behaviors we find in open source projects. However, archetypes can also be prescriptive or aspirational. They collect the types of effects we want to achieve and highlight the characteristics of projects that succeed in achieving those effects.

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, co-development, and community, and how tightly the components of the project are related.

[7] 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.[8]

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.

[8] 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 (but also has characteristics of "Wide Open")*

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. Similarly, the LLVM/clang toolchain is a project where collaboration allows the sharing of costs and resources on non-core business software.

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, in the same or similar roles, 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 less 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"[9].) 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.

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

[9]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."

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.

Single-Maintainer Houseplant

Examples: *All those small packages in npm, Ruby Gems, Python PyPI, and other package management repositories.*

Characteristics: This is a project started by one developer to fulfill an immediate and small-scale personal or professional need[10], and that then stays at that size because that's the appropriate size for what the project does.

This is probably the most well-known and widely written-about type of open source project, because it is the quickest "stable evolutionary state" to reach in open source. Vast numbers of houseplant projects are started, since they're so easy for one person to start — you don't have to coordinate with anyone else. Even though only a small percentage of them gain much adoption, that percentage still accounts for a large proportion of distinct open source projects total. Many such projects stay under the single founder's stewardship for a long time, if the user base doesn't grow too large and there's no technical need for the project to grow significantly in complexity or capabilities.

A Single-Maintainer Houseplant is most likely to transition to another archetype when it succeeds in gaining widespread adoption. Even if its niche is well-defined enough that adoption does not necessarily imply feature growth, the increased rate of incoming bug reports and related inquiries may still overwhelm the maintainer.[11] This is because the rate at which a piece of software receives bug reports is mostly proportional to the size of its user base, not to the size and complexity of the software nor even to number of actual bugs in the code.

To the extent that a Single-Maintainer Houseplant project has a community, it is usually a loose community that forms a "star topology"[12] with the single maintainer as hub. Most communication is between a given contributor and the maintainer — albeit still visible to all — with relatively fewer inter-contributor discussions than happen in other archetypes.

This archetype often overlaps with "Upstream Dependency", as many dependencies also happen to be small, single-maintainer packages.

- **Licensing**: Can be copyleft or non-copyleft (but more often the latter when also an Upstream Dependency).
- **Community standards**: Entirely dependent on the personality of the single maintainer. We have seen a wide range of community standards in this archetype, from

[10] The classic phrase is that such a project starts by "...scratching a developer's personal itch" (*http://www.catb.org/ esr/writings/cathedral-bazaar/cathedral-bazaar/*). This dermatological metaphor has become so well-known by now that it is probably hopeless to try to dislodge it, but we can at least tuck it away discreetly in a footnote.

[11] The burden experienced by unfunded or underfunded individual maintainers of widely-used open source code is one of the threads running through Nadia Eghbal's influential 2016 report *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure* (*https://www.fordfoundation.org/about/library/reports-and-studies/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure*) and the followup discussion it spurred. The report's topic is broader than just single-maintainer projects, however.

[12] *https://en.wikipedia.org/w/index.php?title=Star_network*

welcoming and responsive to almost aggressive non-engagement. Even in when the maintainer is by nature welcoming, however, broad adoption of the software can strain a single maintainer's time and attention.

- **Component coupling**: Little to none. It is one component.
- **Main benefits**: Easy to start; easy make decisions in.
- **Typical governance**: Benevolent dictatorship by founder.

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 provides 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[13]**:** *Rust (present day), Apache HTTPD, Open Stack (but also has characteristics of "Multi-Vendor Infrastructure")*

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.

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

[13]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.

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 self-sustaining, 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 non-technical 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. E.g., 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. When small, this archetype often overlaps with "Single-Maintainer Houseplant".

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[14], 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.

- **Licensing**: Varies, sometimes missing, often incoherent.
- **Community standards**: Community is dormant to non-existent.

[14] 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 *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.)

- **Component coupling**: Depends on the project, but in practice often standalone spaghetti.
- **Main benefits**: Doesn't cost anything to run.
- **Typical governance**: None.

Archetypes At A Glance

	Main Benefit	Main Drawback	Situational Considerations	Dev Speed	Typical Participants	Ease of Onboarding	Community Standards	Typical Governance	Measure of Open Source Success
B2B	Driving industry adoption of your technology.	Little or no collaborative development.	Requires major market power to be effective.	Fast; pace set by business goals.	Organizational reps.	Hard.	Oriented toward organizations.	Founding org, w/ some partner influence.	Adoption by target partners; successful projects built around core project.
Multi-Vendor Infra	Collaboration with partners; address a set of shared problems.	Sometimes off-putting to individual contributors.	Business needs of participants affect community management.	Usually moderate, but depends on needs of participants.	Organizational reps.	Medium.	Welcoming but formal; difficult for individuals.	Committee of organizational reps.	Partner org variety & participation; participant longevity.
Rocket Ship To Mars	Quick, focused effect in a specific area.	Collaboration only available from those who share a very specific vision.	Everything depends on success of original vision.	Fast; escape velocity.	Founding organization.	Hard.	Focused on core group.	Founder governs with iron fist.	Dev speed; adoption by target users; reaching tech goals.
Single-Maintainer Houseplant	Easy to start.	Single maintainer may become burdened by success.	Starts by filling a small niche, then grows.	Medium - fast.	Founding dev and one-off contributors.	Varies.	Varies.	Founder leads.	Either is stable with single maintainer or eventually transitions to another archetype.
Controlled Ecosystem	Can build a sustainable ecosystem in which founding organization has strong influence.	Compromise needed to avoid forks (esp. commercial).	Participants have many motivations (commercial & non-commercial).	Medium.	Founder, some external core contributors, many plugin contributors.	Medium.	Welcoming, with some onboarding structures.	Benevolent dictatorship; tries to avoid forks.	Adoption by target users; extension developers growth.
Wide Open	Large-scale collaboration; community can become self-sustaining.	Effort to maintain onboarding paths & manage all participants.	Differing commitment & engagement levels among participants.	Slow - medium; some process overhead.	Open to anyone; participant demographic depends on project.	Easy.	Very welcoming, formalized onboarding systems.	Group-based; consensus / democratic.	Contributor growth; contribution efficiency; variety in where ideas and decisions originate.
Mass Market	Large user base can make project broadly influential.	Huge user base needs filtering for dev community.	Contributor base does not accurately represent user base.	Slow - medium; swift change destabilizes user base.	Organizational reps, redistributor reps; some users who are technical.	Easy to medium.	Fully open; scales via users helping users.	Main organization leads, with outside input.	User awareness that product is FOSS; non-technical contributor growth; effective filtering of user feedback to devs.
Specialty Library	Ensure quality solution to a specific problem; can lead to new partners.	High barriers to entry; relatively small developer pool.	Standard-setting effects (*de facto* or official).	Gets slower over time, as library stabilizes.	Developers with expertise in the relevant field.	Depends on technical complexity.	High barrier; contributors need to be experts.	Multi-party committer group.	Adoption in intended domain; high quality of contributors and contributed code.
Trusted Vendor	Loyalty of downstream consumers helps project stability.	Primary org must be careful how it uses its position.	Customer needs vs open source project needs.	Medium. Primary vendor momentum vs third-party needs.	Customer reps (both paying and non-paying); some one-off contributors.	Medium to hard.	Clear boundaries: users have mainly roadmap input.	Main vendor leads.	Lack of competitive forks; vendor's leadership accepted by community.
Upstream dependency	Broad reach across (hence insight into) many dependee projects.	Developer base can sometimes be lightly motivated.	Usage patterns of downstream consumers.	Medium; may slow down as standard settles.	Downstream devs.	Depends on technical complexity.	Welcoming; amenable to one-time contributions.	Informal, maintainer-led, committer groups.	Multiple competitive uses; participant longevity; bug reports are technical and constructive.

Contributors and Archetypes: Setting Expectations

Knowing a project's archetype helps to answer an important question:

> *How much, and in what ways, should we invest in attracting, onboarding, and working with contributors?*

Setting contribution expectations accurately is one of the best moves a project can make. For example, an open source project is not obligated to be "Wide Open", but it should avoid accidentally misrepresenting itself as Wide Open if it's not going to provide the kind of contributor support that a Wide Open project would normally provide.

Potential contributors want to see what they're getting into before they invest significant effort. The more clearly a project provides such information up front, the easier it is for suitable contributors to find their way in and, equally importantly, for unsuitable ones to self-select out. Remember that "*contributor*" here does not necessarily mean "*volunteer*": a potential contributor might a salaried employee acting on behalf of some other company, and the impression a project makes on that contributor will be carried back to their team and possibly to their management.

Here are some of the questions developers typically have regarding participation in an open source project:

- I'm considering contributing. What's the typical onboarding path? How far can a third party contributor go?
- I am a new contributor. How much influence should I expect to have in this project, if I continue contributing at my current level? What about if I start contributing more frequently or more substantially?
- I am part of the maintainance team. How much time should I expect to spend mentoring new contributors?

Where each project puts the answers to such questions will vary, but a typical place might be in a `CONTRIBUTING.md` file or in a section called "`Contribution Expectations`" in the project's `README.md`. The exact language will vary from project to project. Here is an example for a project with the "Rocket Ship To Mars" archetype:

```
## Contribution Expectations

 This project is currently focused on high-velocity development toward
 a very specific goal: <summarize goal in one or two sentences here>.

 We welcome patches with bug fixes, and we're happy to answer
 questions if you've already put in some effort to find the answer
 yourself.  Please note, however, that we're unlikely to consider new
```

```
feature contributions or design changes unless there's a strong
argument that they are fully in line with our stated goals.  If
you're not sure, just ask.

Our technical scope and project governance may open up later, of
course, especially after we release version 1.0.  For now, though, we
would characterize this project as being an example of the "Rocket
Ship To Mars" archetype (see <URL here> for details about RStM and
other open source archetypes).
```

Signs of Health: Metrics and Archetypes

The purpose of metrics, in the context of the archetypes, is to tell you whether your investments are having the desired effect.

Metrics are only meaningful when you start from your goals[15]. What are you trying to maximize, what is your investment so far, and are you seeing the kinds of results you want? It's rarely a question of "Are we doing a good job or a bad job?". Rather, it's "Are we doing the job we meant to do, and how can we do it better?"

For example, your goal is to create an ecosystem that spans an industry, with a Multi-Vendor Infrastructure project, then an important metric to watch is what companies are contributing to the project — not just who, but the frequency and depth of their involvement, how often third-party organizations work with each other as peers in the project without the founding organization's involvement, etc.

Compare that with a Wide Open project, where long-term stability is a key goal. There, signals of health would be diversity of contributors, reduction in onboarding time, rate of non-employee contribution, etc. If one of your goals in the project is to use open source involvement to develop a hiring pipeline for new talent, that could affect how you approach onboarding: a broad but shallow pool may be good enough. Broad and deep sounds even better of course, but it also takes more investment and may not be worth the cost.

The point of metrics is not to *identify* the archetype — an archetype is more a target one aims for than a reality one discovers — but rather to tell if the project is succeeding at its goals. A secondary purpose is to get advance notice of forces that may cause the project to evolve from one archetype to another. If you're expecting certain metrics but seeing different ones, it could indicate that the project is transitioning to, or possibly from, some other archetype.[16] Whether this is desirable or not depends on your goals.

Because archetypes overlap, and because a given metric usually applies to multiple archetypes, we treat metrics as their own unified topic here, rather than organizing them by archetype. The combinatorics of looking at every archetype with every metric would be prohibitive, so instead we just mention the archetypes for which a given metric is especially relevant.

[15] See section "Setting Open Source Goals", p. 7
[16] See "Transitions: Changing From One Archetype To Another", p. 37.

These metrics below are in no particular order, and are intended to be used creatively — it might be that, for your particular project, variations of some of these would be most useful:

- **Ratio of one-off inquirers to repeat inquirers.**

  For *Mass Market*, expect very high; for *Wide Open*, expect high.

- **Duration of organizational participation, and "weight" (of contribution and influence) per organization.** Duration might need to be measured relative to market cycles.

  B2B: look for these metrics to be higher for the organizational partners you care most about.

- **Ratio of large change requests to small / one-off change requests.**

  B2B: look for this ratio to be higher than in other types of projects.

- **Ratio of organizationally-motivated changes to individually-motivated ones.**

  This ratio should be higher for *B2B* and *Multi-Vendor Infrastructure* projects than for other archetypes.

- **Existence of customized forks / vendor branches.**

  Long-lived customized forks are more common, and less worrisome, in *B2B* projects than in other types of projects.

- **Rate of conversion of bug reporters to code contributors.**

  Higher in *Specialty Library*, *Controlled Ecosystem*, and *Upstream Dependency* than in other types of projects.

- **Frequency of appearance as topic of conference presentations, hackathons, etc.**

  Higher in *B2B* and *Multi-Vendor Infrastructure* projects than elsewhere.

- **Ratio of lead-organization contributors to external contributors.**

  In *B2B* and *Multi-Vendor Infrastructure*, a high proportion of contributors should be coming from the organizations that are most involved (probably even more so in the former than the latter).

- **Ubiquity of use within a defined industry area.**

  If a *B2B* project is not widely-used in the business area in which it is applicable, it is unlikely to be meeting its goals (unless it has a very unusual goal set).

- **Packaging by OS distributions; frequency of being offered as a pre-defined add-on by cloud vendors.**

  Look for these in *Multi-Vendor Infrastructure*, and perhaps in *B2B* depending on the nature of the project.

- **Migration of functionality from core to plugins.**

  This is a special case of increasing modularization; it should be especially common in *Controlled Ecosystem* projects.

- **Frequency with which a representative from the lead organization ("Benevolent Dictator Organization" / "BDO") participates in discussion, especially with respect to discussions that lead toward a decision.**

  Look at both ratio across all discussions and ratio within each discussion thread. By "discussions" here, we include issue tracker threads.

  For *B2B*, *Rocket Ship To Mars*, *Upstream Dependency*, and *Trusted Vendor*, this frequency should be high.

  For *Controlled Ecosystem*, same, but only for core development, not for plugins.

  For *Multi-Vendor Infrastructure*, if it's very high for one organization, that might be a sign that the project isn't as MVI as you thought.

  For *Wide Open* and *Mass Market*, this metric is not particularly relevant. It could be high or it could be low, but it doesn't necessarily tell you much either way.

- **Organizational diversity of responders.**

  In *Multi-Vendor Infrastructure*, again watch out for one organization doing most of the responding. One use of these metrics is to be able to tell a project — perhaps a project that doesn't want to hear it — that although they are aiming for MVI, they're actually B2B or something else.

  In *Trusted Vendor*, *B2B*, *Rocket Ship To Mars*, *Specialty Library*, expect low.

  In *Wide Open*, *Mass Market*, and *Controlled Ecosystem*, expect high.

  In *Upstream Dependency*, this metric probably doesn't tell one very much.

- **Diversity of participants and contributors along several axes:**

  - Nationality (also, location — not necessarily the same)
  - Native language
  - Profession
  - Gender
  - Ethnicity
  - Paid contributor vs voluntary

These are of special interest to *Wide Open* and *Mass Market* projects. An argument — or perhaps several distinct arguments, depending on the archetype — can be made for their importance to other archetypes too.

- **Number of discrete interactions from a newcomer's arrival to them making a contribution (and thence to becoming a committer, perhaps). Relatedly, how many arrivals drop somewhere along the way, and exactly where.**

  In a sense, this measures speed of onboarding, but with interactions instead of calendar time as the clock tick. We define an interaction as participation in a distinct thread (including an issue tracker thread).

  The reason to measure by interaction instead of by calendar time is that calendar time can vary for reasons that are entirely about the individual and not about the project. For example, someone who is busy and not focused on this project might only interact a few times per year while still eventually getting to the point of being a regular contributor, while someone who has more time or more motivation might do the same much more quickly.

  This *may* be independent of archetype, and it may be the first of a whole class of metrics that are useable as signs of health but that don't necessarily correlate strongly with any particular archetype. We welcome feedback on this question and on the need for deeper discusison of cross-archetype metrics.

Ecosystem Mapping

Ecosystem mapping is a technique for building a shared understanding of how an open source project's stakeholders affect the project.

The best way to make an ecosystem map is to hand-draw it on a large piece of paper or on a whiteboard, as a group exercise. An ecosystem map should be lightweight, messy, and quick, and ideally redone regularly because a healthy ecosystem is never static. We recommend starting with *who* and going from there to *what*, examining at least these elements:

- **Users** *(a.k.a. "end users")*
- **Contributors**
- **Service Providers** & **Support Providers**
- **Partner Organizations** *(e.g., co-investors or funders)*
- **Instances** *(i.e., production deployments)*
- **Competing or Adjacent Projects** *(open source and proprietary)*

Figure 1 shows a simplified real-life example. It is based on an actual map drawn for the Arches Project (*archesproject.org*)[17]. The version shown here is deliberately reduced to just a few entities so it will fit on a page; the real map is much larger and more complex.

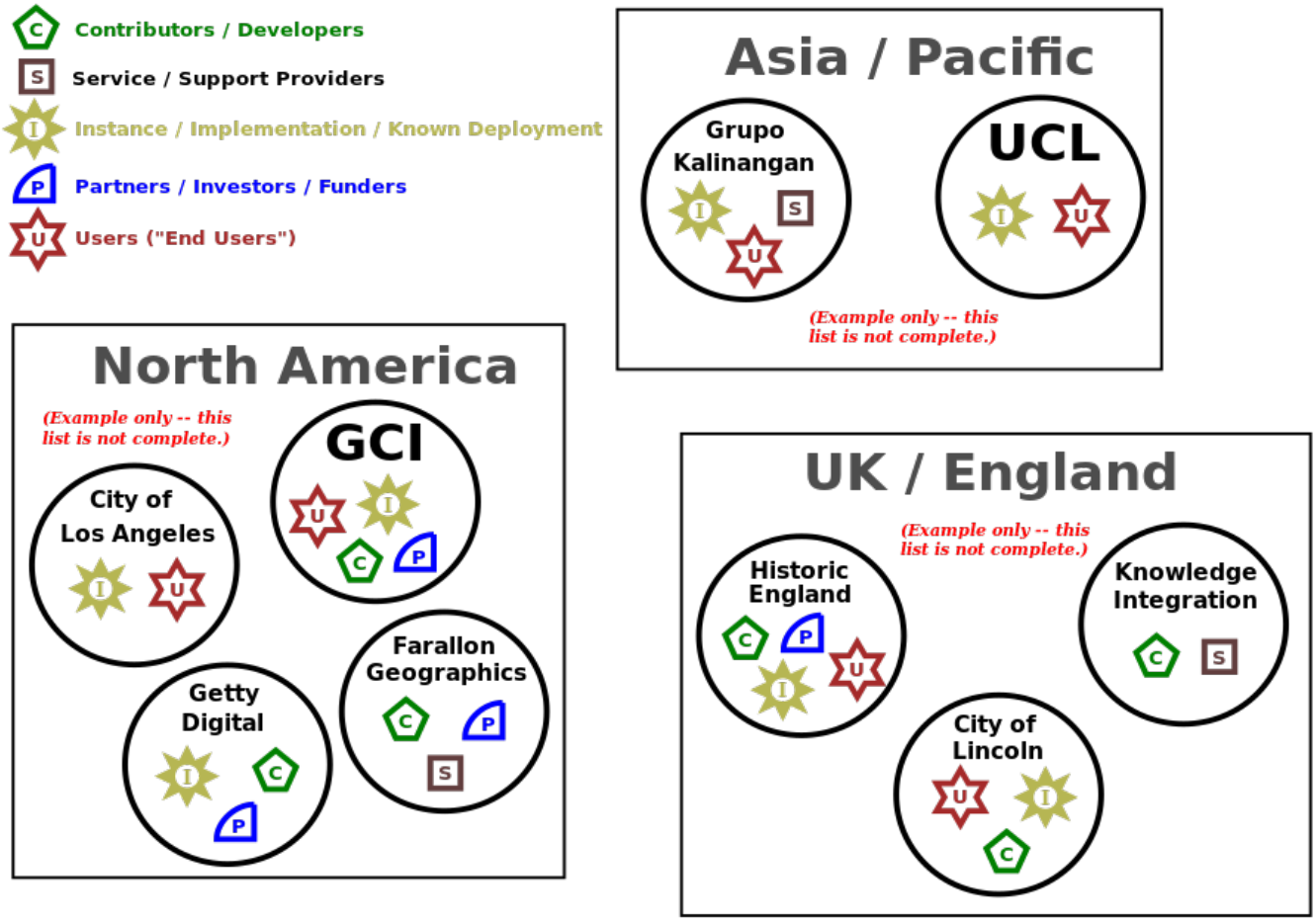

Figure 1: *Example ecosystem map: a simplified and abridged representation of the Arches Project ecosystem.*

Depending on the needs of your project and its archetype, you can tweak how you draw the map:

[17]Some background will help: Arches is an open source platform for managing cultural heritage data, started by two sponsors, the Getty Conservation Institute and the World Monuments Fund. It quickly grew to involve a number of different participants. Some of them are cultural heritage organizations (e.g., Historic England) that both contribute to Arches development and deploy instances of Arches themselves. Others are commercial service providers who deploy Arches on behalf of customers. Still others represent deployments (and thus indirectly those deployments' users) by influencing the project through feedback in the project's forums and participation in user workshops, conferences, etc, more than through direct code contribution. Full disclosure: the Getty Conservation Institute is an OTS client, and we have advised the Arches project since its inception.

- You can use shapes of varying sizes to differentiate between larger and smaller organizations (which could mean larger or smaller in absolute terms, or in terms of the entity's degree of involvement in the project).

- The key categorizations ("Contributor", "Instance/Deployment", etc) may be different for your project. The ones offered here are just suggestions, though they are ones we have found work decently well across many projects.

- You might draw lines of communication or cooperation between different nodes, and those lines could use different styles or colors to represent different types of connections.

  Seeing the relationships and dependencies between different entities in the ecosystem can be just as important as categorizing the entities. When depicting major vendors and significant clients, for example, you could draw lines to indicate who is servicing which vendors in which markets and niches. (We didn't include such lines here only because it would have cluttered up the example diagram too much.)

- This example map happens to show geographic regions, because they were important for that particular project, but not all ecosystem maps need do so. Other groupings might be more useful for other purposes — for example grouping by application domain, or linguistically rather than geographically.

How to Use an Ecosystem Map

Sketch the diagram as a whole first, being as inclusive as possible, and then step back and look for patterns. As with drawing the map in the first place, this interpretation step is best done as a group exercise.

For example, if you notice that entities of a particular type – say, small companies, or entities located in a particular geographic region – offer service and support but are *not* contributors, that raises the question of whether the project could do more to help them participate. If there are many such entities and they share a linguistic background, that could signal that the project should consider investing in translating its documentation into that language, or even hiring developers who know that language.[18]

An ecosystem map is usually tuned to a specific purpose. For example, the map in Figure 1 was drawn mainly to help understand *who* the participants in that ecosystem are and what their motivations are. By contrast, the map shown in Figure 2 (p. 35) was drawn

[18]It might also be a good idea to look to see if those entities have already created their own language-specific forum in which they discuss the open source project — in other words, your open source project may have grown without your knowing it yet. In such cases, the value of investing in some bilingual developers or translators to help bring the two groups closer together is greater. OTS's report "OpenDRI and GeoNode: A Case Study On Institutional Investments In Open Source" (*https://opendri.org/wp-content/uploads/2017/03/OpenDRI-and-GeoNode-a-Case-Study-on-Institutional-Investments-in-Open-Source.pdf*) talks more about the importance of inter-lingual connection in open source (p. 37).

more to clarify the flow of information within the Tor project. It does not list external collaborators or deployments individually, nor does it show geographical regions. Its value lies in giving collaborators an overview of all the interest groups around the Tor project and how they relate to one another, by showing which ones are "nearer" and "farther" from each other in terms of communications and concerns. It's a quick way to help answer the question "Have we thought of every type of participant who might care about what we're proposing?"

It is not unusual to make several different maps of the same ecosystem in an afternoon, each map emphasizing a different view. Don't try to cram everything onto one map: the result will be too confusing to look at. Just make several maps. In a fast-moving project, the terrain will shift often; ecosystem maps should be made quickly and used immediately.

Ecosystem Maps and Archetypes

Ecosystem maps help a project fully realize all the benefits available from its archetype(s), mainly by making ambient knowledge explicit and thus usable.

For example, in a Multi-Vendor Infrastructure project, there might be a set of organizations known to be using the software (known by their activity in the project's discussion forums) but who are not well-represented among the project's contributors. An ecosystem map can reveal patterns about those organizations. They may have things in common — geographically, linguistically, or technically — that the project can take advantage of to become more inclusive.

On the other hand, in a Trusted Vendor or Upstream Dependency project, an ecosystem map might reveal cluster of users that can be solicited for input about the project's feature roadmap. Depending on the size and nature of the project, some of those users may be small vendors selling commercial support for the software. You might vaguely know of one or two such vendors, someone else on your team might know of another, and so on. When you get together and draw a single ecosystem map, that knowledge gets combined and made actionable. Suddenly you see all the vendors, or most of them anyway, and have the ability to make a coordinated plan for working with them.

Ecosystem maps are especially useful during archetype transitions (see section "Transitions: Changing From One Archetype To Another", p. 37), planned or otherwise. In fact, the exercise of drawing an ecosystem map is sometimes what first makes your team aware that the project's archetype is changing. If you've been trying to run a Wide Open project, say, but your ecosystem map makes it clear that your organization (or perhaps some other organization) wields decisive influence and is structurally positioned to continue to do so, you may decide that Controlled Ecosystem is a better fit for the reality of the project — and being able to acknowledge that openly as soon as possible is usually a good thing for all participants.

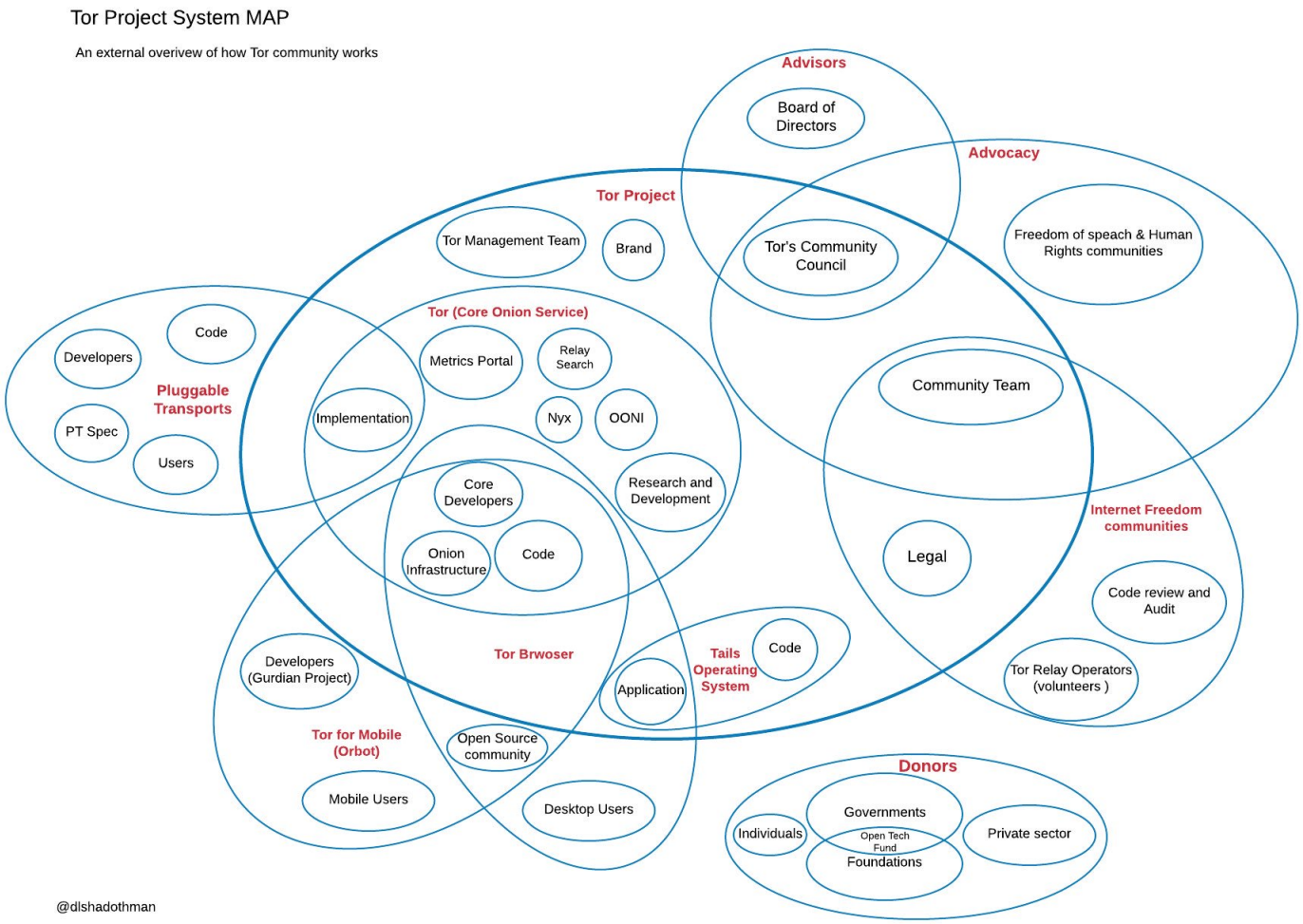

Figure 2: *Sample ecosystem map for the Tor project.*

Business Models and Archetypes

A project's archetype is just one component of some larger strategy, and the strategy should drive the archetype, not the other way around. To think "Okay, we have a Rocket Ship To Mars project here. What business models can it support?" would be putting the cart before the horse. The better way is to start from strategy: "The business model we see this project being part of is X. What archetypes work well with X?" (There may be more than one business model, of course, just as a project may end up incorporating elements from multiple archetypes.)

Different archetypes suit different business models, similarly to how different open source licenses suit different kinds of for-profit activity. In licensing, there are times when a non-

copyleft[19] license is the right choice, given your intended business model (perhaps widespread adoption is crucial to your plans), and there are times when the strongest possible copyleft[20] is the right choice (perhaps avoidance of proprietary derivatives is crucial to your plans).

Similar considerations apply when choosing an archetype. If gaining mindshare among individual developers is crucial, then you need an archetype (such as Wide Open) that prioritizes participant onboarding, even if it means sacrificing development speed or other technical goals. If building a coalition to forestall market domination by a large competitor is your goal, then an archetype designed for inter-organizational collaboration (such as B2B or Multi-Vendor Infrastructure) is a better choice.

Think of "business model" as a superset of "revenue model". A revenue model is just one part of a business model. You must start from a clear definition of your product and your value proposition. For example, in proprietary software, businesses that look at first glance like they are based on a royalty-per-copy revenue model often turn out, on closer examination, to be more complex than that. Sometimes a business *thinks* it is selling per-seat or per-server usage rights, while its customers think they're what they're paying for is support, vetted installation, configuration convenience, and regular security and feature updates.

Most revenue models are compatible with open source, in general. Really, the only one that is incompatible with open source is the one based on per-copy royalties for software sales. While that can be a very successful[21] revenue model under certain circumstances, it is far from the only one.

This report is not the place for an in-depth discussion of value creation, value capture, market definition, and revenue models. We note these complex topics merely to point out that any given open source archetype will be better-suited to some business models and worse-suited to others. In order to determine which archetype is best for your project, you must start from the high-level strategy — in which revenue is just one element — that motivated your involvement in project in the first place.

[19] A.k.a. "permissive". See "Elements that vary across open source licenses", p. 42.

[20] E.g., Currently the AGPL. As of this writing in early 2019, the debate over whether the newly-created "Server Side Public License" from MongoDB is or is not a free software / open source license is still very new. Because it has not yet been evaluated by the OSI nor declared a free software license by the FSF, we take a conservative approach and do not consider it here. This does not imply any position on whether it is or isn't a free and open source (FOSS) license. It simply acknowledges the practical reality that SSPL-licensed software is not going to be widely treated as FOSS unless and until the relevant certifying organizations treat it as FOSS. See "Appendix A: Basics of Open Source Licensing", p. 42, for more context.

[21] If you define success primarily in terms of revenue generation, that is. For mission-driven organizations, success involves more than that anyway.

Transitions: Changing 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 that) during its initial push to a 1.0 release to a mode much closer today to Mass Market.

An example of a less centrally-driven and more incremental transition is the WordPress blog platform and content management system, 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 has occurred with the acquiescence and probably the encouragement of WordPress's primary commercial backer, Automattic, Inc. 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, although that does not mean that every such transition is consciously 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 parties 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 toward 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.

Choosing An Archetype: Practical Questions

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 can use, modify, and redistribute the code under that license.

That's all that is absolutely necessary for open source. Everything else — governance, project management, contribution policy, marketing, long-term roadmap, etc — 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, in governance of the particular master copy from which your organization makes 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[22].

  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.

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

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[23]. 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.

  - 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?

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

- 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 "Appendix A: Basics of Open Source Licensing" (p. 42) for more on this topic.

  - 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", p. 42).

  - 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", p. 45.)

- **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?

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.

Appendix A: 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.[24] 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

[24] 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.

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 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[25], 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 if SaaS product includes GPL'd code, then 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 more recent variant of the GPL, the Affero GPL (AGPL), partially 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 turn makes it difficult for anyone

[25] 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.

else to charge a lot of money for a copy, so in practice the price of acquiring copies is driven to zero pretty quickly.[26]

- *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 someone 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 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,

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

even when that device natively runs an unmodified version of the code — a version that, because it is unmodified, can pass the signature check.[27]

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.

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).

[27] 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.*

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).

Appendix B: 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*.

Copyright

For the most recent version of this report, please see *https://opentechstrategies.com/archetypes*.