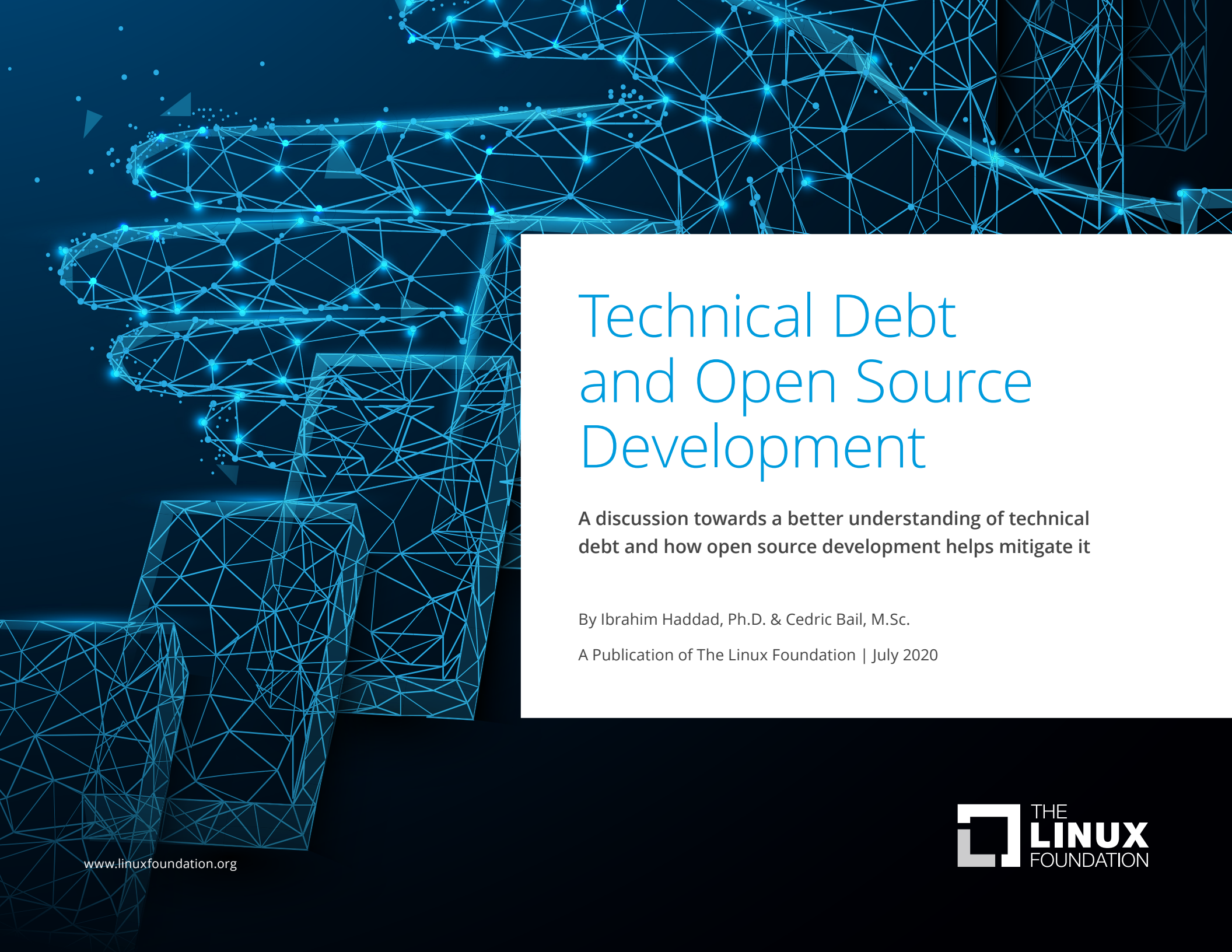

Technical Debt and Open Source Development

A discussion towards a better understanding of technical debt and how open source development helps mitigate it

By Ibrahim Haddad, Ph.D. & Cedric Bail, M.Sc.

A Publication of The Linux Foundation | July 2020

www.linuxfoundation.org

THE LINUX FOUNDATION

Abstract

This paper grew out of a phone conversation the authors had on a Sunday morning mid–March 2020 while being confined in their homes. Both authors worked together within the Open Source Group at Samsung Research and directly experienced minimizing internally carried technical debt via working with upstream open source projects. That experience covered dozens of open source projects used across multiple products and business units with varying degrees of involvement and experience with upstream development. The paper provides an overview of the problem of technical debt at large scale. It includes discussions on identifying technical debt, how to minimize it, the role of open source development, and strategies to address the issue.

Disclaimer

The opinions expressed in this paper are solely the authors' and do not necessarily represent their current or past employers' views. The authors would like to apologize in advance for any error or omission and are open to feedback and updates.

Table of Contents

Technical Debt

Definition

Technical debt, a term used in software development, refers to the cost of maintaining source code that was caused by a deviation from the main branch where joint development happens. A wider interpretation of what constitutes technical debt is proprietary code by itself:

- It has been developed by a single organization.
- It is source code that the organization alone needs to carry and maintain.
- In some cases, the organization depends on a partner's ability to maintain the code and carry that said debt.

A noteworthy clarification is that upstream code is not without technical debt when the upstream project doesn't have resources/time to maintain itself via its developers' community. An example of this scenario is the various companies that depended on the OpenSSL project without contributing to the project and realizing that the project was maintained only by a single person during their spare time. This was the specific case scenario that motivated the Linux Foundation to launch its Core Infrastructure Initiative to support open source projects critical to our modern infrastructure.

Symptoms

How would you identify the symptoms pointing to the existence of technical debt? And, what are these symptoms? In this section, based on our experiences, we list several such symptoms with a brief description of each one of them. This is not intended as an exhaustive and comprehensive list but rather a list of the most common and widely observed technical debt symptoms.

- **Slower release cadence** Time increases between the delivery of new features
- **Increased onboarding time for new developers** Onboarding new developers becomes highly involved due to code complexity where only insider developers are familiar with the codebase. The second manifestation of this symptom is the difficulty in retaining developers or hiring new developers.
- **Increased security issues** At least, experiencing more security issues than the main upstream branch.
- **Increased efforts to maintain the code base** Maintenance tasks become more time consuming as the body of code to maintain becomes larger and more complex.
- **Misalignment with the upstream development cycle** illustrated in the inability to maintain pace, be aligned, with the upstream development and release cycles.

Types of Technical Debt

There are several types of technical debt. We are not aware of a standardized or commonly agreed-upon definition to describe these various technical debt types. Therefore, in this section, we present our interpretation.

Temporary Technical Debt

A team may be working on a complex feature that possibly touches several components, systems, or subsystems. The need arises to carry out certain technical debt for a temporary period as things get developed and integrated with the upstream branch. We are aware of the fact that we're creating technical debt. However, our purpose is to accelerate product development, and the end goal is to repurpose and merge the fork with the upstream branch at a later point.

Unknown Technical Debt

Unknowingly creating technical debt as a result of bad engineering practices. An example of this scenario is poorly-written code that is not accepted into the upstream branch and is not even a candidate for reusing somewhere else. We're stuck with this code.

Purposely Created Technical Debt

This unusual type of technical debt is being created on purpose. An example of such a case would be an organization that wants to maintain certain features exclusive to them without sharing them with the broader community. As a result, such organizations find themselves creating this fork to keep it independent without merging it with the upstream branch. Over time, such forks grow, leading to larger technical debt and increased associated maintenance costs.

Obsolete Technical Debt

Obsolete technical debt is a unique use case of technical debt resulting from the "not invented here" syndrome or the result of isolated development of new components that could have benefited the broader community. However, due to a lack of technical oversight or leadership, one organization only uses that development. Simultaneously, the world moved on to solve the problem and created a defacto solution (or a standard) that is now incompatible with what the organization has developed. This situation makes that development a source of technical debt by obsolescence.

Organizational Technical Debt

It is widely discussed how the source code that an enterprise creates often matches that enterprise's organization -- a very interesting theory. In some cases, it so happens where code should be developed and where it ends up being developed do not match. When developers cannot push back on managers and have the code written by the right people, or they can't contribute to the right piece of code, the result is often a piece of duct tape on code that shouldn't be there and that nobody wants to deploy. This is technical debt.

The Many Causes of Technical Debt

A large number of factors contribute to the creation and growth of technical debt. In this section, we explore the most common causes and provide a brief description of each one.

- Low-quality code that can't be upstreamed for some reason, such as it doesn't meet the code quality criteria set by the target open source project. Another manifestation of this factor is what is commonly referred to as "spaghetti code."

- Self-serving code that is only useful to the specific company contributing the code without much use to the general community. Such code (or functionality) is usually not accepted upstream, and the recommendation is often to adjust it in such a way to benefit general use cases and the wider number of users.

- Fragmented development leading to duplicated efforts and competing implementations

- Lack of effort to drive code upstream -- in many cases, the team's organization creating the code does not have enough bandwidth to work with the community and drive the code into being accepted in the upstream branch. This may improve effectiveness in the short term but creates technical debt and long term negative consequences in code maintenance and upkeep.

- Intrusive code that requires additional coordination across multiple components or various systems/ subsystems.

- Time to get code accepted upstream that causes temporary technical debt until the code is accepted and merged in the upstream branch. It has no adverse effect on the long term.

- Lack of testing preventing complete test coverage and causing failure in submission.

- Lack of documentation or documentation that is not up-to-date.

- Poor technical leadership and inadequate engagement with the technical community lead to being sidelined and having to, later on, play catch up with the rest of the world as they move on.

- Ongoing change in requirements within the internal development efforts in any given organization.

- Non-standard technology or lack of alignment with a given standard.

- Organizational obliviousness that combines the aspects of poor technical leadership and unawareness of the technical direction of upstream development. This scenario is increasingly common in non-digital-native companies who are increasingly forced into development work.

Consequences

Creating and carrying technical debt will have several negative effects on development efforts, including:

- The higher cost of code maintenance.

- Slower innovation and development cycles.

- Paying interest on the debt -- payment of technical debt is in the form of additional development needed to keep up with the main branch, the competition, and the rest of the world.

- Possibly missing on new features in the main branch or having to backport all new development into the forked branch internally.

- Duplicate work with the main branch arising due to the delta between the internal and public branches being too large.

The worst possible consequence is the effect on the long term maintainability of the code base where organizations often find themselves maintaining their own fork.

How does technical debt accumulate?

Arthur Bloch is an American writer, author of Murphy's Law books. He is quoted, "Friends come and go, but enemies accumulate." This is a great quote to reference when discussing technical debt as just like enemies, technical debt accumulates. How does it happen? Figure 1 offers an illustration of a basic scenario that explains how technical debt gets created and carried over.

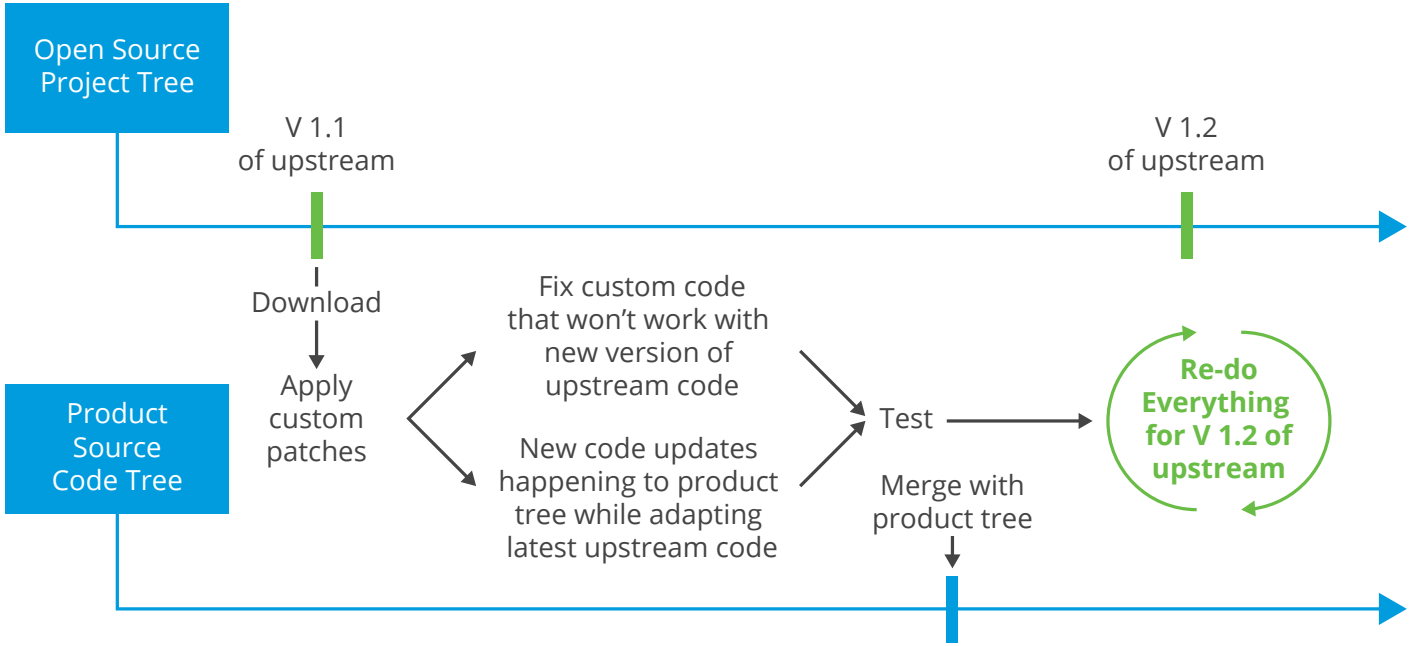

Figure 1: Development cycles without upstream integration

Working with Technical Debt

Identifying Technical Debt

Every line of code is potentially technical debt that consumes engineering time best put on specific business needs. Figuring this out is mostly about figuring out your business goals and what your team is putting time on. Answering a few questions can help with this thought exercise:

- What is your team working on?
- What do you need to tell a new hire regarding your software stack?
- How often can you update your full software stack?
- Do you know all the software you depend on?
- What usually breaks?
- What dependencies are the most painful to deal with?
- What would have been the alternative for every single component of the software stack used?
- How active are the communities of each of these alternatives?
- How much work is needed to maintain that component if the community disappears or gets stagnant?
- How much effort to switch to this component?
- How hard is it to debug problems reported by your customers or engineers?

This series of questions will help you start discussing technical debt and where your blind spots are.

Minimizing Technical Debt

The core question to address now is how to minimize technical debt and minimize its impact on development efforts.

Choice of programming language

The programming language or development framework used to build the product poses certain restrictions on your developers. The higher the complexity of a language or a framework, the harder it is to maintain a workforce that can work with it. It becomes a balance between the constraint of the system you are building your software for and the access to the necessary developers base to get the job done. In general, for most cases, you are better off with a higher-level[1] language as it will:

- Facilitate hiring developers that provide good results.
- Help third parties identify and solve problems.
- Be portable and easily maintained due to community involvement.
- Be more tolerant of mistakes and allow for a simpler solution.

1 The choice of a high-level programming language is context specific, however, as examples of such languages, we'd like to suggest Go, Python, C#, and Typescript.

- Important online source of documentation, tutorial, and pre-made solutions.

Choice of ecosystem

Your application is always built on top of a software stack of language, framework, and operating system. This is typical of the structure or composition of the ecosystem the application is going to live in. This ecosystem will shape the technical debt, and developers should know how it aligns with their own goals. For instance:

- Linux distributions have different terms of support and guarantee various levels of API/ABI/security over time.
- The choice of programming language impacts the ability to run software over time. It affects the ability to execute code as the runtime and build environment it runs within evolve. Also, it might pose some limitations on your ability in finding new developers to manage those changes.
- Modern language and framework also tend to bypass Linux distributions to package software. This factor might have a long term impact on your software in the same way the Linux distribution will, so it should also follow your API/ABI/security needs.

Choice of dependencies

As the world moves forward, more open source software of high quality is made available. This is a continuous process, and it is essential to evaluate each piece of software that your organization is developing. These dependencies simplify the necessary work and enable them to build more complex features and solutions. Still, they can also become technical debt if their communities are not healthy enough. Choosing the right dependencies and contributing to the important ones to you reduces your technical debt by sharing it with the world.

Also, what was true for the value of dependency a few years ago might not be anymore. This means that like dealing with technical debt, evaluating your dependencies needs to be done continuously.

Example 1: The Role of Custom Build Systems

Maintaining an operating system is more than a full-time job. It requires making sure that

- It works reliably and securely over time,
- Building each component is reproducible over time,
- Each component is properly evaluated and tested,
- Licenses are respected, and,
- That you have a robust and secure update mechanism.

This type of work is undervalued as putting together a Linux environment can now be done in a matter of hours; however, this is just the first step of maintaining an operating system over a long period. Today, with the ARM entrance in the server market, it is possible to have a more easily standardized distribution for embedded devices. Debian, Ubuntu, Redhat, and SuSE, to name a few, provide ARM server distributions that run just fine, or with a small tweak, on any embedded device and remove the need for maintaining a custom operating system and associated package build. It also provides developers with standardized tools to transfer knowledge from one environment, the cloud, to the embedded market. Finding developers that can now work on embedded systems becomes easier as hiring managers can tap into the larger market -- the Linux server market.

It is most likely that an important trend in embedded device development is going to be in picking an ARM server distribution with some kind of Long Term Support, along with a small service written in a higher-level language, much in the same way that the cloud industry writes them. Python, Node.js, and Go all have a bright future in the embedded systems industry. The next time you have an embedded Linux project, consider going with a standard Linux distribution that provides some form of Long Term Support and reduces your future technical debt.

Example 2: User Interface Framework

It is easy to gather a few open source components and run them on a Linux Kernel to name this assemblage a Linux Distribution. It is not very difficult to display a picture on the screen with some text and call it a small UI framework. And in the same way that maintaining a Linux distribution is a never-ending task, writing and maintaining your UI framework will also be a never-ending task. Consider the need to display language from around the world correctly, the need for accessibility, the need to scale up and fit a more constrained environment, and support different rendering systems as the world moves to better technologies. There is no end to maintaining any UI framework. You can easily observe these in the development of existing UI frameworks. Qt, GTK, and EFL are more than 20 years old today. They required hundreds of developers to get where they are, and we should expect them to require the same level of effort for the next 20 years. React, and ReactNative require hundreds of developers as well, a language change isn't changing the need to address all this external constraint. When you pick a UI framework, understand that you choose a community, and rely on them to carry its technical debt. Being able to step in and help might be necessary to ensure that this community stays healthy and keeps moving that debt off your shoulder.

It is also recommended to pay attention to the licenses in effect and generally who owns any given project's IP assets. Depending on a UI Framework and without being involved in its development nor paying any licensing fees, you are inherently weakening one of your core dependencies if you are making a visual application and so increasing your technical debt. It is usually hard to switch frameworks, and the more complex your application, the harder it becomes to change frameworks. As for Linux distributions, you'd want to be involved in some form with your upstream dependencies to align with your own needs in the long run. Contributing to upstream can take many different forms, and you should choose the one that matches your business the best (code, monetary, documentation, marketing, etc.)

The Role of Upstream Development

It is expected to branch out or fork and do your development as long as the end goal is to contribute back to the upstream branch. Figure 2 illustrates this process.

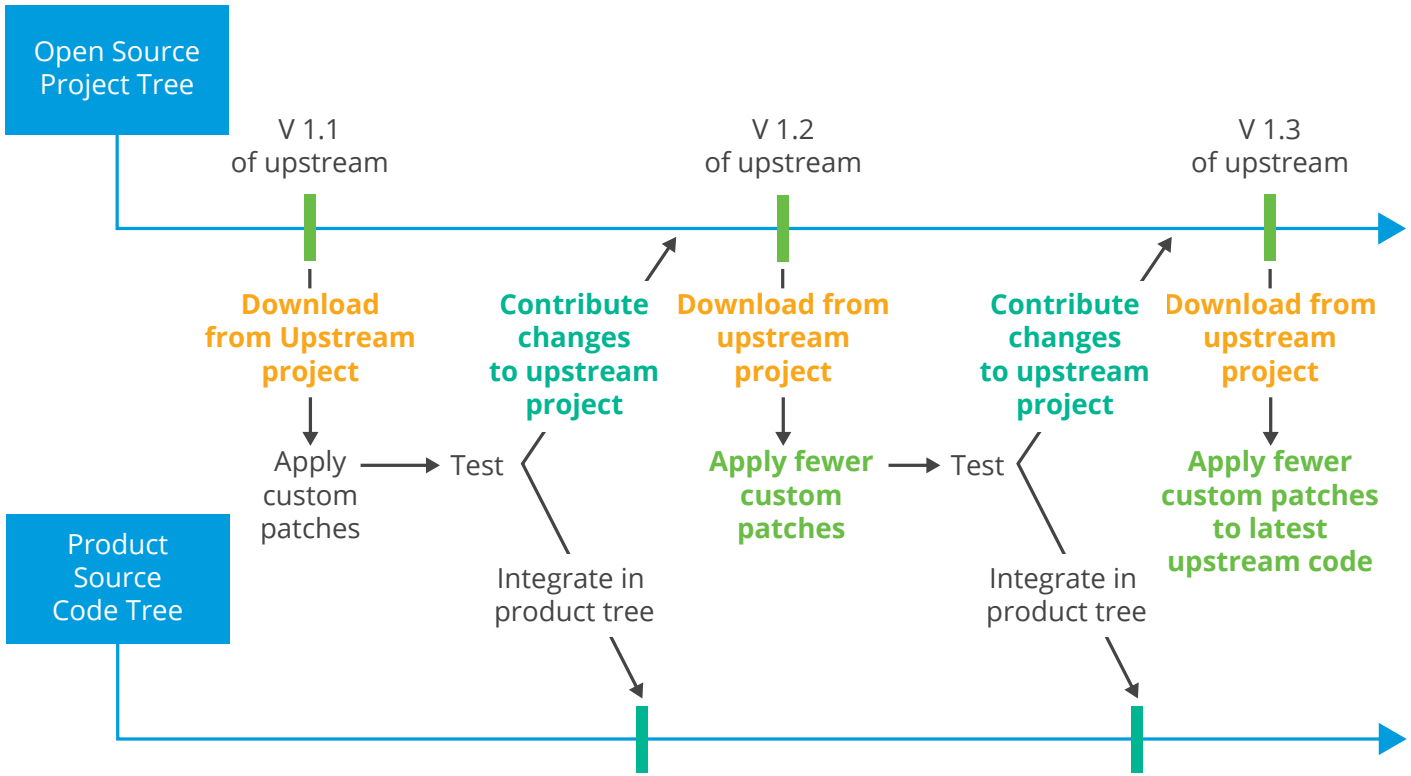

Figure 2: Development cycles with upstream integration

- **Continuous integration and continuous delivery/ deployment:** Features are available more quickly to developers; new code appears in developer trees sooner. At a higher level of quality, each code commit requires testing, regressions, and bugs are more visible.
- **Release early and often:** The release early and often practice has numerous proven advantages. The "release early" allows others to provide feedback and participate in the development, welcoming new ideas that can be incorporated.

Simultaneously, code is still flexible and offers time for problems to be flagged by others before development gets too far. On the other hand, the "release often" makes it possible for the codebase changes to be easier to understand, debug, and drive to maturity, and at the time, facilitates the ability to maintain the rapid pace of development and innovation.

- **Peer review:** Share as early as possible, even during the design phase of your code; Requests for comments are expected; subsystem/maintainer model with multi-layer hierarchy; by the time code is released, it has typically been reviewed many times. Code is always reviewed before being committed. Enables projects to accept code from a much wider range of contributors (establishes a web of trust).
- **Ongoing or continuous testing:**
  - Early discovery means faster triage and fixes.
  - Smaller changes make troubleshooting easier.
  - Regressions are noticed earlier.
  - Helps subsystem maintainers determine which code submissions to accept.
- **Projects may have multiple build and test cycles.**
  - Build service tools can automate the process.
  - Typically tightly aligned with feature freezes.
- **Easier Maintenance** (changing technology, security fixes)
- **Better testing, bug report gathering, and analysis**
- **Focus on modular designs and architecture:** Modularity has several benefits: it allows projects to scale, minimizes contention over common code with a smaller core and features implemented as plugins reduce collisions, creates a natural separation of tasks and scope, allows features to be available more quickly to developers, and if well done modular designs often have fewer interdependencies with clear interfaces.

Upstream - Unifier of Efforts

Done well, developing in alignment with upstream is a guarantee that you will have little to zero technical debt. However, that requires involvement and active participation in upstream open source projects. You can not enforce that upstream is going in the direction that benefits you. Still, you can influence it by contributing to it and sharing your goal/will with upstream and influencing other contributors to understand your needs.

Addressing Technical Debt at Scale

If your organization uses hundreds or thousands of open source packages and makes modifications to many of them, you will need to consider a contribution strategy that will allow you to contribute code back to the core or essential components to minimize technical debt towards such strategic components. What are some of the ways to address the technical debt on a large scale? We explore two sets of practices that help that at policy and process level and then at the development level.

At the Policy and Process Level

- Blanket approval to contribute for dedicated open source developers.
- Faster approval path that allows contributions to upstream projects to flow much faster.
- Flexible IT support enables developers with needed tooling (now facilitated with the Windows Subsystem for Linux and virtual machines if you can't afford a dedicated Linux environment for your developers).
- Structure performance reviews that reward developers who follow the set processes/policies and work towards minimizing technical debt
- Guaranteed time for your developers to work with the upstream project

At the Development Level

- Explain your business goals to your engineers by sharing your vision and goals. In the end, they are the ones implementing it, so they should know what you have in mind.
- Ensure that your developers and new members joining your team understand what technical debts are and why you choose to maintain the technical debt you already have.
- Have realistic expectations in terms of merging your contributions with the upstream branch.
- Embrace the review/feedback cycle. There will be multiple back-and-forth cycles as part of the review process with the upstream development.
- Don't be too selfish with your contributions. Get involved with upstream on tasks that are not necessarily directly needed in the short term by your organization, but improve upstream viability and health.
- Encourage your developers to explicitly leave comments in the code when they are adding technical debt.
- Work for the short term but plan for the long term.

Too late! Technical debt is already here. What should we do?

The organization you work with has accumulated a lot of technical debt, and all the symptoms are there. Now, what should you do about it? We wish there was a silver bullet, but there isn't. The approach you take will depend on several factors; however, you can start examining these different options:

- Choose what feature/functionality needs to be saved.
- Identify the code that is still useful.
- Remove code that should not be maintained or used anymore.
- Reduce the need for branches/fork.
- Refactor, clean and upstream the code that can be upstreamed.

These activities are time-consuming and will slow down your development cycle as you dedicated developers to focus on this effort. It will also be hard for the organization to add any features during that time and might be very controversial or counter to the ongoing efforts.

It is also possible that there exists now an open source project that does or could provide the feature or some of the features you need. Migrating to it might also make more sense than trying to deal with the current codebase. It is something not to forget, the world moves even when you are not looking, and there might be something out there now that does what you need. Do not let your organization pride prevent you from looking at other solutions and use the one that makes most business sense.

Another more radical approach: Give up on your technical debt and drop the code altogether. This can be done in multiple ways. If your business can not afford to maintain the code still and that there aren't enough clients to justify its existence, just drop it. For that, be explicit with your clients and tell them that this code is going to be deprecated and sunsetted. This is the equivalent of declaring bankruptcy on your technical debt and ceasing payment on it.

Recommended Practices

In this section, we provide a list of recommended practices listed in random order. Please keep in mind that they may not all apply to your specific situation and organization as you read this section. Some may apply, some may not. Therefore, it is essential to do a self-evaluation of an appropriate practice to adopt within your company and possibly how such presented practices can be adjusted to provide the best possible outcome for you.

- Adopt an upstream first philosophy.
- Careful evaluation for any custom code that is not going upstream.
- Always plan to merge back with the main branch. Forks and side branches are ok as long as there is a plan to merge with upstream.
- Align internal development effort with the upstream branch release cadence
- Allow fast approvals for upstream contributions. This is done via a clear and lightweight policy and process to facilitate interactions with upstream developers.
- Update your performance metrics to incorporate metrics related to technical debt as part of the overall performance goals to ensure that development goals are not achieved due to a high or unacceptable technical debt cost.
- Train developers/managers to identify and mitigate technical debt scenarios.
- Require all code to be properly documented to be better understood by upstream reviewers and most likely will contribute to a faster acceptance cycle (Document also why code is not upstreamed)
- Follow the release early and often practice. Don't build a huge code base and then decide to upstream it. Share design, early code and submit large contributions in smaller, independent patches that build on each other.
- Track the code you choose to not upstream: Re-evaluate at every opportunity. If you spot an upward trend in the internal maintained code's size, this calls for a discussion and reevaluation of prior decisions.

Conclusion

Open source has a significant role, and aligning your development efforts with upstream open source projects can result in a direct positive impact on the amount of the technical debt an organization carries. Just as financial debt involves paying interest, technical debt has a different kind of interest that needs to be carried: It's not interest-free!

In many cases, technical debt is unavoidable short term. Carrying technical debt is mostly a decision that developers need to make all the time. The long term goals of any engineering effort should be to minimize and eliminate technical debt resulting from any development effort. With proper policies, processes, training, and tooling, organizations can help mitigate and guide the engineering efforts towards lowering technical debt.

Feedback

Suggestions for improvement will be appreciated. Please send comments to the authors directly.

About the authors

Cedric Bail (M.Sc.) is a senior software engineer currently contracting with EASi specializing in embedded Linux. Cedric has worked as a software engineer for various embedded Linux platforms at a mobile operator, an Internet service provider, and, most recently, one of the largest consumer electronics companies. His focus has always been to optimize software for constrained environments, improving their API, and introducing new components as needed in upstream software to match business requirements.

Twitter: **@cedric_bail**
LinkedIn: **linkedin.com/in/cedricbail/**
Email: **cedric.bail@free.fr**

Ibrahim Haddad (Ph.D.) is the Executive Director of the LF AI Foundation. Haddad has held technology and portfolio management roles at Ericsson Research, the Open Source Development Labs, Motorola, Palm, Hewlett-Packard, the Linux Foundation, and Samsung Research.

Twitter: **@IbrahimAtLinux**
Web: **IbrahimAtLinux.com**
LinkedIn: **linkedin.com/in/ibrahimhaddad**
Email: **ibrahim@linuxfoundation.org**

The Linux Foundation promotes, protects and standardizes Linux by providing unified resources and services needed for open source to successfully compete with closed platforms.

To learn more about The Linux Foundation or our other initiatives please visit us at **www.linuxfoundation.org**