THE LINUX FOUNDATION | Research

Getting Started with Open Source Development

A Practical Guide for New Contributors

Ibrahim Haddad, *Ph.D.*
Head of Infotainment Engineering, Volvo Cars

Foreword by
Shuah Khan, *The Linux Foundation*

September 2026

In partnership with

LFX Mentorship

Getting Started with Open Source Development

Maintainers ***own a project's upstream repository*** and decide what gets merged.

Contributing to a project means releasing that work under its ***existing license***.

Permissive and copyleft licenses place ***very different obligations*** on downstream users.

A Developer Certificate of Origin is a lightweight ***attestation*** added to each commit.

Starting with already familiar tools ***shortens the learning curve*** for new contributors.

Recent commit activity and fast issue responses signal a ***healthy, welcoming project***.

Small, focused pull requests fixing one clear issue are easiest for maintainers to merge, and each commit should stay focused on a ***single, clear, logical change***.

Documentation, triage, and ***translation*** are contributions maintainers welcome as much as code.

Trust in open source is earned through ***repeated, reliable contributions*** over time.

Syncing a fork with upstream often prevents small conflicts from becoming large ones.

LFX Mentorship is a structured mentorship program that pairs new contributors directly with experienced maintainers across major open source projects.

To secure your employer's approval for open source contributions, propose a ***narrow, time-boxed project on an existing production dependency*** to reduce maintenance and deliver ROI.

Table of contents

Foreword

We all know open source humming along smoothly is vital to the world around us. Bringing new people into projects in the open source development model is voluntary, unlike the recruiting and hiring model of closed source. Open source projects that are effective in attracting and retaining talent clearly outline how their development process works, providing documentation detailing their processes for sending contributions for review and inclusion. Furthermore, they ensure their documentation clearly states effective and ineffective methods to engage and collaborate with them. While the documentation is the first, they bolster it by providing passive training as part of the development and review process.

These steps alone aren't sufficient to attract and retain talent. It is often hard for new developers to figure out how to go about choosing a project that is right for them. Technology might narrow things a bit, but still there are several things to take into account. After sending their first contribution, new developers have to follow through with continued contributions and engagement to gain credibility. There is no clearly identified path to becoming an expert or a maintainer. People who stay for the longer term have the mindset to adapt and work with uncertainty to advance the project towards a shared and mutually beneficial direction.

Mentorship is an important avenue that shortens that path, easing the journey for new developers to connect with experts and experts to train them. Structured programs pair new developers with maintainers who already know a codebase inside out, and webinars or interactive sessions let experts with limited time reach more newcomers than one-on-one guidance ever could. None of this guarantees that a mentee stays in the same project, or in open source, indefinitely. Developers move on to other projects and other careers. What they carry with them is the knowledge and the habit of contributing in the open, and that outlasts any single mentorship.

This report lays out the fundamentals-first path: how open source projects work, what their licenses and contributor agreements actually require, how to choose a project worth the effort, and how to turn a first contribution into a lasting role in the community. It is a practical starting point for anyone ready to move from using open source to building it.

Shuah Khan
Linux Fellow,
The Linux Foundation

Executive summary

Open source is how most modern software gets built, and the projects behind it depend on a steady supply of new contributors. That supply never arrives through traditional hiring. The development model is voluntary, so every project's health rests on people choosing, without being asked, to show up and do the work. For a developer, learning to contribute is one of the fastest ways to build real experience and join the communities that maintain the tools already in daily use.

The skills built along the way are the real payoff of the work: reading unfamiliar codebases, collaborating asynchronously across time zones, and earning the trust of maintainers who owe a first-time contributor nothing. These capabilities transfer directly into any engineering career and are difficult to build anywhere else at the same pace. Each merged contribution also adds to a public track record that a hiring manager, a client, or a future maintainer can inspect directly, without needing anyone's introduction.

This report is written for first-time contributors. It builds the mental model required before any code is written, explains the licensing and contributor-agreement rules that govern participation, walks through the mechanics of a contribution on every major platform, and shows how a one-time contributor grows into a trusted member of a project.

The path is incremental by design: start small, learn one project well, and let responsibility grow with the track record. The judgment and the standing that come from doing this well take longer to build than the mechanics do, and they are what make the investment worth making.

Introduction

Most organizations now build on open source, and a growing number contribute back as strategy rather than goodwill. That shift created lasting demand for people who can work within open source projects. Contributing teaches skills not easily acquired elsewhere. Individuals learn to read large unfamiliar codebases, collaborate asynchronously with people across time zones, defend a design decision in public, and ship to a quality bar set by maintainers who have never met them. There is a practical payoff too. A public contribution history is a portfolio anyone can inspect, backed by a set of references who can speak to the contributor's work. None of it requires permission. It requires knowing how to start.

Learn the ground rules before writing code

This section covers what to understand before any code is touched, from how a project is organized to the licensing and sign-off rules that govern a contribution.

How an open source project works

A project lives in an upstream repository, the canonical home of the code. Maintainers are the people with authority to accept changes and cut releases. Contributors propose changes, which maintainers review and either merge or decline often with a request to make some modifications to the source code to make it upstream eligible.

All of this process happens in the open. Discussion sits in issues, proposals sit in pull or merge requests, and decisions sit in review threads anyone can read. This structure is worth understanding before forking anything, because it shapes the entire workflow: a contributor works on an independent copy and proposes changes back, and accountable maintainers evaluate each one. The contributor's task is to make that evaluation easy.

Licenses and the obligations of contribution

Every healthy project carries a license, and projects that use an **OSI-approved license** are the safer choice. The license is the legal basis of the project, and it sets the terms under which a contribution is released.

Two families of licenses matter on day one. Permissive licenses such as MIT, BSD, and Apache 2.0 let others use the code with few obligations, including inside closed source products. Copyleft licenses such as the GPL family require derivative works to carry the same terms, which keeps the code and everything built on it open.

Neither family is better in the abstract. What matters is that contributing to a project means releasing the contribution under that project's license. We recommend new contributors to read the license and have a basic understanding of it before they start contributing to the project.

Contributor agreements: DCO and CLA

Many established projects require a formal step before merging a contribution, and a new contributor tends to encounter one of them sooner than expected.

The Developer Certificate of Origin, or DCO, originated in the Linux kernel community. It is a lightweight attestation: a **Signed-off-by: Full Name <email@example.com>** line added to each commit, generated automatically by running **git commit -s**, certifying that the contributor has the right to submit the work under the project's license. Many projects enforce this with an automated check that blocks any pull request missing the sign-off.

A Contributor License Agreement, or CLA, is a heavier legal document signed once, often through a bot triggered by the first pull request. It grants the project or its steward the rights needed to redistribute and, in some cases, relicense contributions. Some CLAs are individual; others require the contributor's employer to sign.

A first contribution blocked by a failing DCO or CLA check is being blocked for exactly this reason. Reading the project's contributing file and following it resolves the check.

Choose the right project

Knowing the ground rules only helps once a project has been chosen to apply them to. This section covers what starting actually requires in time and confidence, how to find a project that fits a contributor's skills, and how to read its health before committing time to it.

Time, capacity, and confidence

There are two questions that will come before any technical ones: how much time does contributing to projects actually take, and whether a newcomer into an open source project has anything worth offering. Neither of these questions has a simple and definitive answer.

There is no correct number of hours. A first contribution can come from a single evening spent updating documentation, and a sustained habit can run on a few hours a week. The second pattern holds up better over time. For a contributor without employer support, that slot has to come from personal time, and the practical approach is to treat it like a habit rather than a project: the same day, the same hour, a small and repeatable scope of work. A quiet Sunday morning or a lunch break spent reviewing one issue accomplishes more, over a year, than a weekend occasionally set aside for a large push that keeps slipping. A contributor whose employer might support this instead has a different case to make, and that case is addressed later in this report.

The second barrier is the sense that a project's existing contributors are too experienced to need help from someone new. The Linux Foundation Research's **Mentorship in Open Source** found that nearly two-thirds of participants entered the program lacking confidence in their ability to contribute, before any assessment of their actual skill. That gap is common, documented, and rarely about ability. Every maintainer started as an outsider to the codebase and built their way up. For new contributors, the work that supports and keeps a project healthy, such as improving documentation, testing, writing test cases, or being a code janitor hunting for bugs, is a great starting point.

Start from existing context

With millions of projects available, the hard part is finding the right fit. Starting from tools and languages already in use is the most efficient path, because understanding the software as a user shortens the learning curve considerably. Interest carries weight too, since a first contribution demands patience, and curiosity is what sustains it.

Read the health signals

Not every project is a good place to begin. Ten minutes spent reading the signals before investing time pays off. Recent commit activity indicates the project is alive. The response time on open issues and pull requests indicates whether maintainers are present. Issues labeled **good first issue** or **help wanted** indicate that the project actively wants newcomers, and the tone of recent review threads reveals whether the community is a place worth spending time in.

A project that answers contributors within days beats a more famous one where pull requests sit untouched for months.

Say hello before committing

Most projects run community channels: a mailing list, a chat workspace on Slack, Discord, or Matrix, a forum, or discussion threads on the host. A brief introduction, some review of recent conversations, and a question about where a newcomer can help go a long way. Maintainers will often hand a good first task directly to a contributor who makes contact this way.

Some organizations formalize this on-ramp rather than leaving it to chance. The Linux Foundation runs **LFX Mentorship**, which pairs new contributors with maintainers across projects such as the Linux kernel, CNCF, Hyperledger, and the Open Mainframe Project. Linux Foundation Research's report **Mentorship in Open Source** found that nearly two-thirds of participants entered the program lacking confidence in their ability to contribute, and 90 percent reported increased confidence by the time they finished. A structured mentorship is not a requirement for a first contribution, but it is a documented alternative for a contributor who wants more guidance than a cold introduction in a chat channel provides.

Set up to contribute

Once a project has been chosen, a little setup stands between the contributor and a first commit. This section covers the accounts and platforms involved and the documentation that specifies how the project wants contributions made.

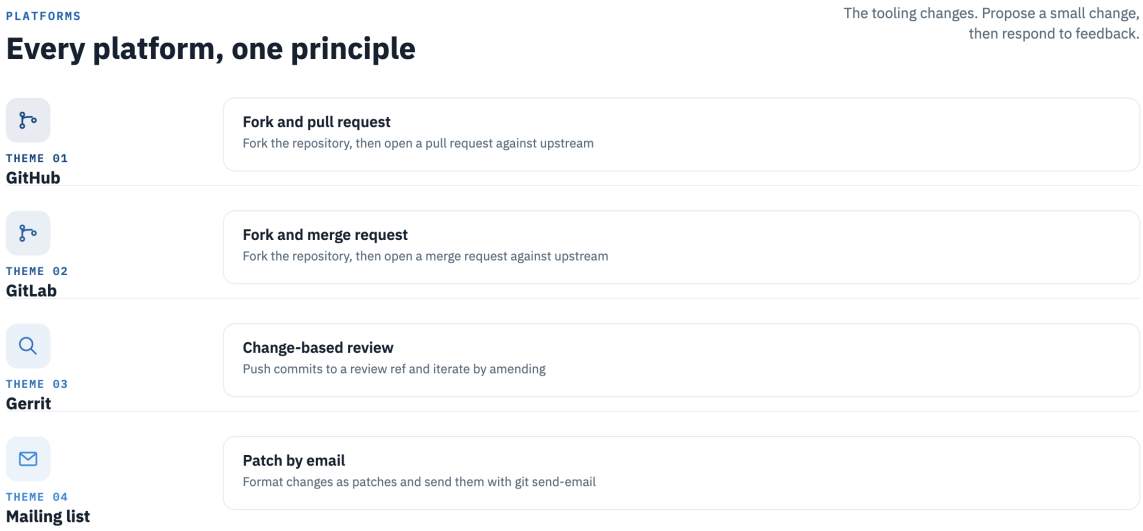
Accounts and platforms

Open source does not live on one platform. GitHub is the largest host and a reasonable default, so a **GitHub account** and basic fluency with its **documentation** are worth having early. Other workflows exist as well, and the project you choose may not be hosted on GitHub.

GitHub uses forks and pull requests. GitLab uses forks and merge requests, the same idea under a different name. Gerrit uses a change-based review model in which commits are pushed for review instead of opening a pull request, a pattern common in large infrastructure projects. Mailing-list projects, including the Linux kernel, take patches sent by email using **git format-patch** and **git send-email**.

Mastering all of them is not necessary. What matters is recognizing that a non-GitHub project is normal and reading its contributing guide instead of assuming the GitHub flow applies.

FIGURE 1
How GitHub, GitLab, Gerrit, and mailing list projects handle review.

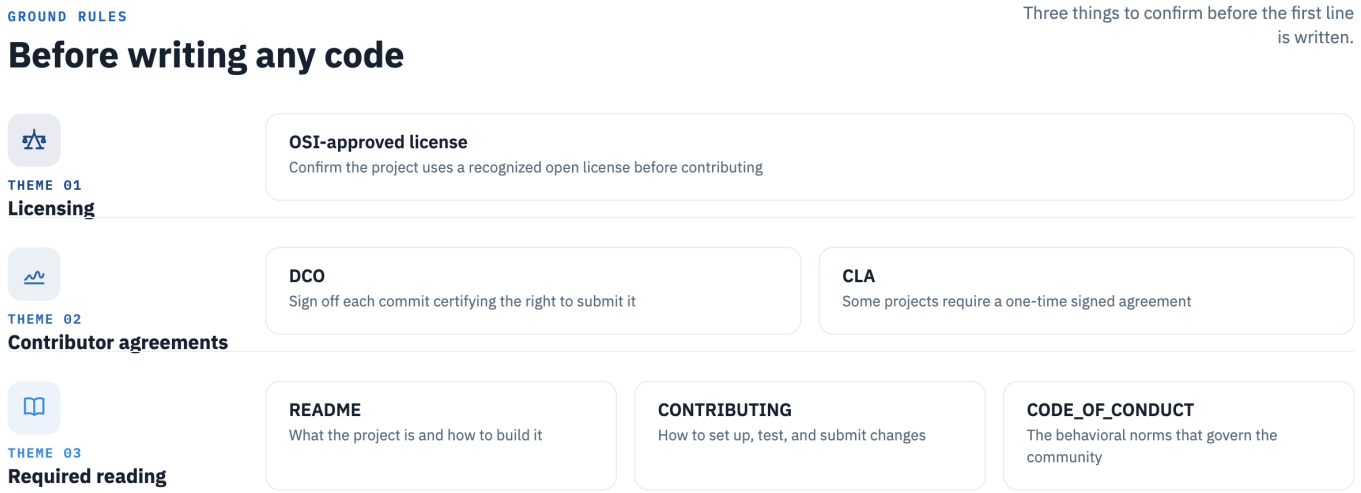


Read the docs that govern contribution

Every well-run project documents how to contribute, and these documents are worth reading before any code is written and submitted for review. The README covers what the project is and how to build it. The CONTRIBUTING file covers how to set up, test, and submit changes, including any DCO or CLA requirement, and, on a growing number of projects, a stated policy on AI-assisted or AI-generated contributions that ranges from a disclosure requirement to an outright ban. The CODE_OF_CONDUCT covers the behavioral norms a contributor is agreeing to, and a GOVERNANCE file, where one exists, covers how decisions get made and who makes them.

Following these guidelines is the single biggest factor in whether a first contribution lands. A change that ignores the stated process creates work for maintainers and signals that the rules went unread.

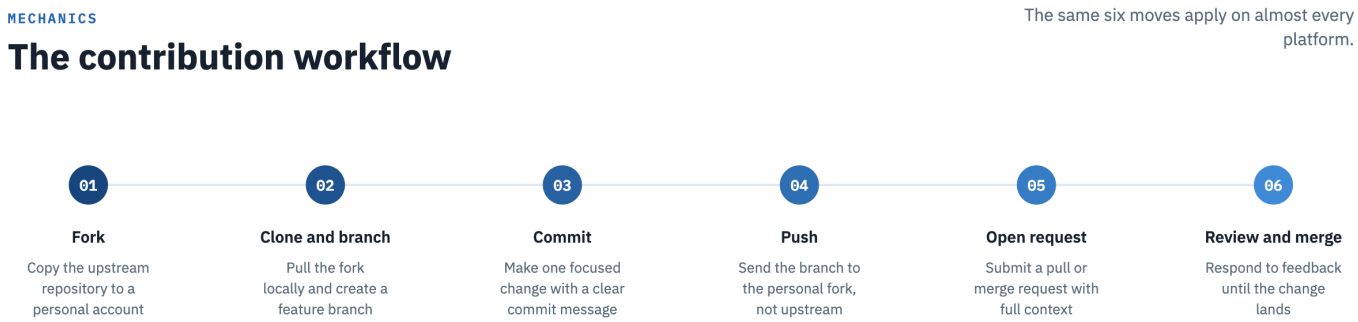
FIGURE 2
The license, contributor agreement, and reading checklist before a first contribution.



Make the first contribution

With tools ready and the project's rules read, the change itself becomes possible. This section walks through finding a small piece of work, preparing it with fork, clone, and branch, and opening the pull or merge request.

FIGURE 3
The contribution workflow, from fork to merge, in six steps.



Find and claim a small piece of work

It helps to start small. A documentation fix, a failing test, a minor bug, or an issue tagged for newcomers makes an ideal first contribution. Before writing anything, a search of existing issues and pull requests confirms the work is not already done or in progress. If an issue is open and unclaimed, a comment stating an intent to take it is standard practice. For anything beyond a trivial fix, opening an issue to discuss the approach first avoids hours spent on a change the maintainers will decline.

Fork, clone, and branch

The standard distributed workflow has a few setup steps a change depends on. Forking the upstream repository to a personal account produces a fully controlled copy, and cloning that fork locally makes it possible to work on it.

The original repository is then added as a remote, named **upstream** by convention, to allow its changes to be pulled in later. A feature branch is created for the change rather than working on the main branch, which keeps the work isolated and the main branch clean.

Forking allows free experimentation without touching the original project. The feature branch keeps separate contributions separate and makes the eventual review cleaner.

Write the change with good commit hygiene

Each change should stay focused on one logical thing. A pull request that fixes one bug is far easier to merge than one that fixes a bug, reformats three files, and renames a function on the side. A clear commit message follows a standard shape: a short summary line (“Fix null check in parser”), a blank line, then a brief explanation of what changed and why. Where the project requires a DCO, **git commit -s** provides the sign-off.

Open the pull or merge request

The branch is pushed to the contributor’s own fork rather than to upstream. New contributors lack write access to the upstream repository, so pushing there simply fails. From the fork, a pull request on GitHub or a merge request on GitLab targets the upstream project. The description should give reviewers what they need: what the change does, why, how it was tested, and a link to the issue it resolves.

Other workflows in brief

Projects that use Gerrit require the commit to be pushed to a review ref, with iteration handled by amending the same commit. Mailing-list projects require the change to be formatted as a patch and sent to the list with **git send-email**, with review answered by sending revised versions. The principle holds across all of them: a small, well-described change is proposed, and feedback is addressed as it arrives. Only the tooling differs.

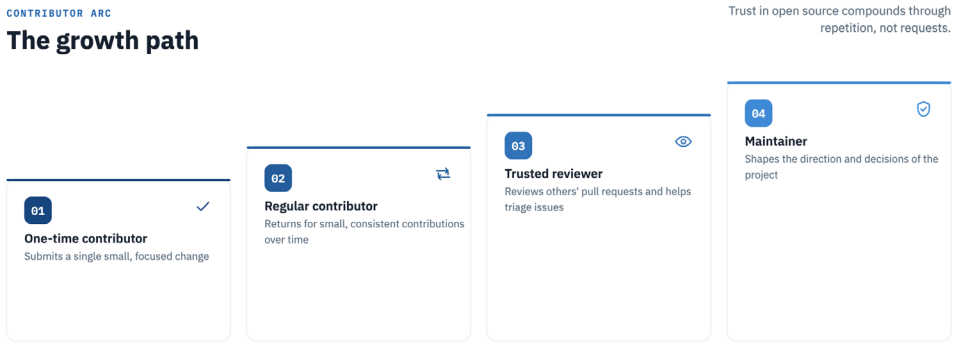
Work with the community

Submitting a change opens a conversation that runs through review and beyond. This section covers how to communicate well in issues and reviews, and how a contributor's standing grows with each contribution.

Communicate well in issues and reviews

These communities are global and asynchronous. An instant answer is rare, and the people reviewing the work are usually volunteers or busy engineers. Staying concise and specific, and bringing context, makes review faster for everyone. A bug report with reproduction steps is worth ten that say only "it doesn't work." Good faith should be assumed, and extended in return.

FIGURE 4
The path from a single pull request to a maintainer role.



Treat review feedback as the process working

Review comments are how open source holds its quality, and almost every first pull request receives change requests. That is the system working as intended. Making the requested changes, asking questions when a comment is unclear, and pushing updates to the same branch so the request updates in place, keeps the process moving. Maintainers remember contributors who are easy to work with, and that memory is how larger responsibilities get earned.

Build a track record

Trust in open source is earned through repetition. Each solid contribution, each helpful review comment, each well-formed issue builds standing within the project. With time, maintainers may invite a contributor to review others' work, grant triage rights, or offer a maintainer role. All of it is earned through the work itself, and offered rather than requested.

Contribute beyond code

Code is one path, and for many newcomers it is not the easiest entry point. Projects need documentation, often the most welcomed contribution a newcomer can make. They need issue triage, confirming bugs and applying labels, which directly lightens maintainer load. They need testing, translation and localization, design and user-experience work for projects with interfaces, and people answering questions in forums and chat.

These contributions are first-class. They teach the project from the inside, put a contributor in contact with maintainers, and frequently lead to code contributions once the codebase is familiar.

Stay in sync with upstream

While a change is in progress, the project keeps moving underneath it. Keeping the fork current with upstream avoids conflicts and rework. Once the upstream remote has been added, **git fetch upstream** downloads its latest history without touching working files. Those changes are then integrated, either by merging with **git merge upstream/main** or by rebasing the branch on top of them with **git rebase upstream/main**.

A caution the basic advice usually skips: rebasing rewrites branch history and produces a cleaner result, but it should be understood before it is used on shared branches. On a personal feature branch, merging is the safer default when in doubt. Either way, syncing often rather than letting a fork drift pays off, because a small conflict resolved today is easier than a large one resolved at merge time.

Sustain momentum and grow

Getting one contribution merged proves the process can be done. This section covers how to turn that single success into a habit and, over time, a larger role in the projects that matter to a contributor.

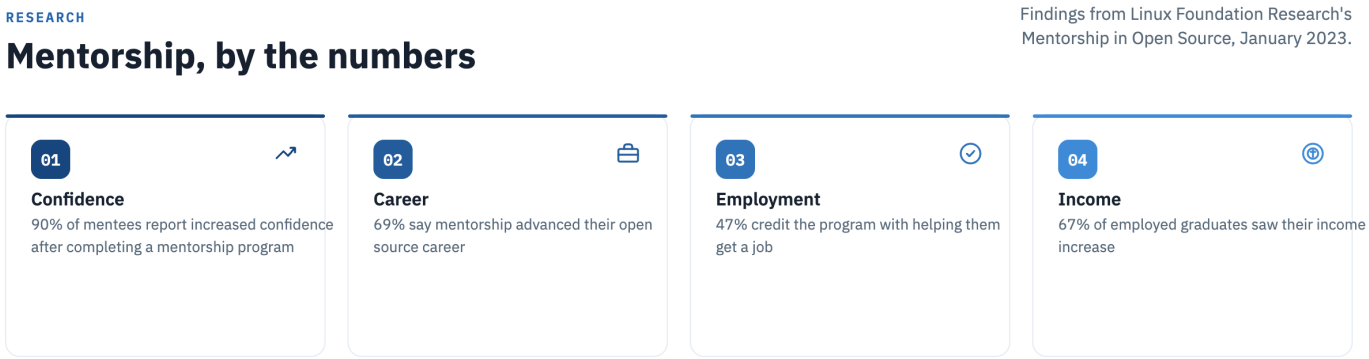
A single merged contribution is a beginning. The contributors who get the most from open source are the ones who come back. Choosing a project worth caring about and making a habit of small, regular contributions is what sustains the arc. As the codebase becomes familiar and the community becomes familiar with the contributor, larger work follows: a feature, a refactor, or reviewing newcomers' pull requests. That is the path from drive-by contributor to trusted regular, and from there to the maintainer responsibilities that shape where a project goes.

The career value compounds along the way. A contribution history stays visible to anyone evaluating the work, the maintainers involved become professional references, and the community becomes a network built around shared technical interest. All of it is built by showing up consistently and doing good work in public.

Linux Foundation Research's **Mentorship in Open Source** found the same compounding effect among formally mentored contributors: 69 percent reported career advancement tied to their participation, 47 percent credited the program with helping them get a job, and 67 percent of those employed afterward saw their income increase. The path there differs, structured mentorship rather than self-directed contribution, but the underlying pattern holds. Consistent, visible work in an open source community produces career outcomes that are measurable, not just anecdotal.

FIGURE 5

LFX Mentorship's confidence, career, and income outcomes for its graduates.



Get support for your open source contributions

A contributor who wants to get an approval from their work for their open source efforts will face a specific conversation with their manager. Most managers default to caution when the request sounds open-ended. The strongest version of this pitch proposes something narrow, time-boxed, and tied to work the team is already doing.

The Linux Foundation Research's **ROI for Open Source Software Contribution**, based on a 2025 survey of organizations, found that contribution delivers a 2 to 5x return on investment, with code contribution specifically returning 3.6 times its cost. The same study found that nearly half of organizations maintain private forks of the open source components they depend on, consuming more than 5,000 labor hours per release cycle and costing an average of $670,000 a year in workarounds for misaligned roadmaps. Contributing organizations report faster security response, roadmap influence, 10 percent faster product development on average, and easier talent retention. The cost of not contributing is real. It is simply carried quietly, inside the team's own maintenance burden.

That case translates into a concrete ask. The strongest starting point is a dependency the team already relies on in production, since a fix there reduces work the team is already doing, not work invented for the pitch. A modest, fixed time allocation, a few hours a week or a portion of a sprint, is easier to approve than an open-ended commitment. A first small deliverable, one merged pull request or one triaged issue, gives the manager something concrete to point to before the conversation about expanding it happens. This mirrors the approach the rest of this report recommends for the contribution itself: start small, produce something real, and let the track record make the case for more.

A few points worth raising directly in that conversation with line management:

- The dependency already runs in production, so an upstream fix removes a maintenance burden the team already carries.
- A landed pull request, a merged security fix, or a roadmap comment is concrete, reportable evidence, not abstract time spent.
- Private patches and local workarounds carry an ongoing cost of their own, one that contributing upstream can reduce over time.
- Linux Foundation Research found that organizations contributing to open source see 10 percent faster product development, easier talent retention, and a 2 to 5x return on the investment.

Conclusion

Open source rewards developers who learn its rules and take part in good faith. The mechanics are learnable in an afternoon. The judgment and the standing take longer, and they are what turn a first pull request into a lasting role.

What would a practical start look like for a new contributor?

- Choose one project already in use and read its README, CONTRIBUTING, and CODE_OF_CONDUCT.
- Confirm whether it requires a DCO sign-off or a CLA, then find an issue labeled **good first issue** and comment to claim it.
- Fork, clone, and branch.
- Make a small, focused change, and sign off the commit if required.
- Push to the fork.
- Open the pull or merge request with a clear description.
- Answer the review and the incoming feedback.
- Rework, get it merged, and repeat.

The skills gained along the way working with the open source community are real and transferable, compounding with every open source project you contribute to and participate in. They also come with membership in something larger: a global community of creators and doers who built the software the world now runs on.

References

Perlow, Jason. *Mentorship in Open Source: Exploring the Intrinsic, Economic, and Career Value of Mentorship Programs.* Foreword by Julia Lawall. The Linux Foundation, January 2023. **https://www.linuxfoundation.org/research/mentorship-in-open-source**

Boysel, Sam, and Adrienn Lawson. *ROI for Open Source Software Contribution: Insight from the Open Source ROI Survey and Economic Model*. Forewords by Chris Aniszczyk, Masato Endo, and Hillarie Prestopine. The Linux Foundation, February 2026. **https://www.linuxfoundation.org/research/contribution-roi**

LFX Mentorship **https://lfx.linuxfoundation.org/tools/mentorship/**

Acknowledgments

The author is grateful to Hilary Carter, SVP of Linux Foundation Research, and Shuah Khan, Kernel Fellow at The Linux Foundation, for reviewing this report and for the suggestions that sharpened it. Special thanks goes to Douglas Byers, an aspiring open source contributor, who was kind to review the report and provide feedback that led to covering additional topics and expanding other areas. The Linux Foundation Research team deserves equal credit: their support runs through every stage of this report, from the first outline to the published version.

Feedback

This report is meant to be useful to developers making their first open source contribution, and it improves with input from the people who use it. Comments on any gap, error, or step that did not match a reader's experience are welcome and can be **shared** with the author directly.

Disclaimer

This report is provided for educational purposes. Licensing, contributor agreements, and governance terms vary by project, and the specific terms of any project under consideration should be reviewed independently. References to specific platforms and tools are for illustration purposes only.

About the author

Ibrahim Haddad, Ph.D., is Head of Infotainment Engineering at Volvo Cars, leading engineering for the company's next-generation in-vehicle software platform. Prior to Volvo Cars, he served as Vice President of AI Strategic Programs at the Linux Foundation, leading LF AI & Data and growing it into the largest neutral open source AI ecosystem, and concurrently as Founding Executive Director of the PyTorch Foundation. At Samsung, he served as Distinguished Engineer and VP of R&D.

Haddad's career spans executive and technical leadership roles at Volvo Cars, Samsung, the Linux Foundation, Ericsson, Motorola, Palm, and Hewlett-Packard. He has championed open source driven by the belief that collaborative development is a faster, better, and cheaper path to innovation. He is a recognized expert in open source license compliance and a long-standing advocate for OSPOs as the structural backbone for external R&D engagement.

Haddad has authored 7 books, 23 e-books, and more than 150 publications covering open source strategy, AI governance, engineering leadership, and ecosystem development. He has advised the World Bank Group and presented to the European and UK Parliaments on open source and AI. He earned his Ph.D. with honors in Computer Science from Concordia University, where he was awarded both the J. W. McConnell Memorial Graduate Fellowship and the Concordia University 25th Anniversary Fellowship.

LinkedIn: **https://www.linkedin.com/in/ibrahimhaddad/**
Portfolio: **https://www.ibrahimatlinux.com**

THE LINUX FOUNDATION | Research

Founded in 2021, **Linux Foundation Research** explores the growing scale of open source collaboration, providing insight into emerging technology trends, best practices, and the global impact of open source projects. Through the use of project databases and networks, and a commitment to best practices in quantitative and qualitative methodologies, Linux Foundation Research is creating the go-to library for open source insights for the benefit of organizations the world over.

x.com/linuxfoundation

facebook.com/TheLinuxFoundation

linkedin.com/company/the-linux-foundation

youtube.com/user/TheLinuxFoundation

https://github.com/linuxfoundation

To reference this work, please cite as follows: Ibrahim Haddad, "Getting Started in Open Source Development: A Practical Guide for New Contributors," foreword by Shuah Khan, The Linux Foundation, September 2026.