

Red Hat

Open Source Development Course

Project Lifecycle & Dependency Management

Marek Čermák <cermakm@redhat.com>



Project release lifecycle

Releases indicate the development status and record the development progress

In order for users and contributors to navigate in the project lifecycle and development status and also for the developers to be able to manage the project, versioning and releases are one of the most important things to do when developing a software.

Different people have different needs

“““

As a contributor to a project that is new to me, I always struggle to **navigate** in the contribution guidelines.

Should/Can I even contribute? **How do I contribute** – is there a specific procedure? When is the right time?

A contributor

“““

When **choosing a project** I want to depend on, I usually look at its development status and decide based on the maturity of the project. Concise documentation is a **+**.

A developer

“““

A project becomes harder to maintain as it grows. The key to success is having a **well-prepared continuous delivery and integration pipelines**. As a benefit, it protects you and your team from an absolute chaos.

A developer & maintainer

What is a “project”?

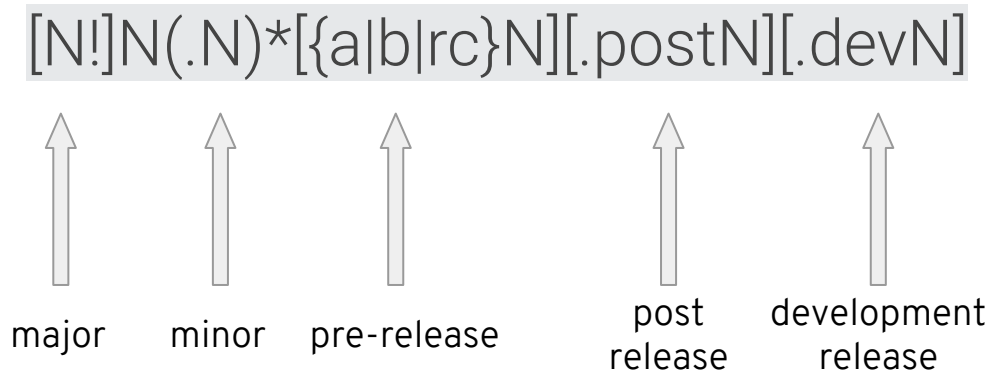
The following definition applies to a Python project, but can be easily translated to other languages:

"Projects" are software components that are made available for integration. Projects include Python libraries, frameworks, scripts, plugins, applications, collections of data or other resources, and various combinations thereof. Public Python projects are typically registered on the [Python Package Index](https://pypi.org/).

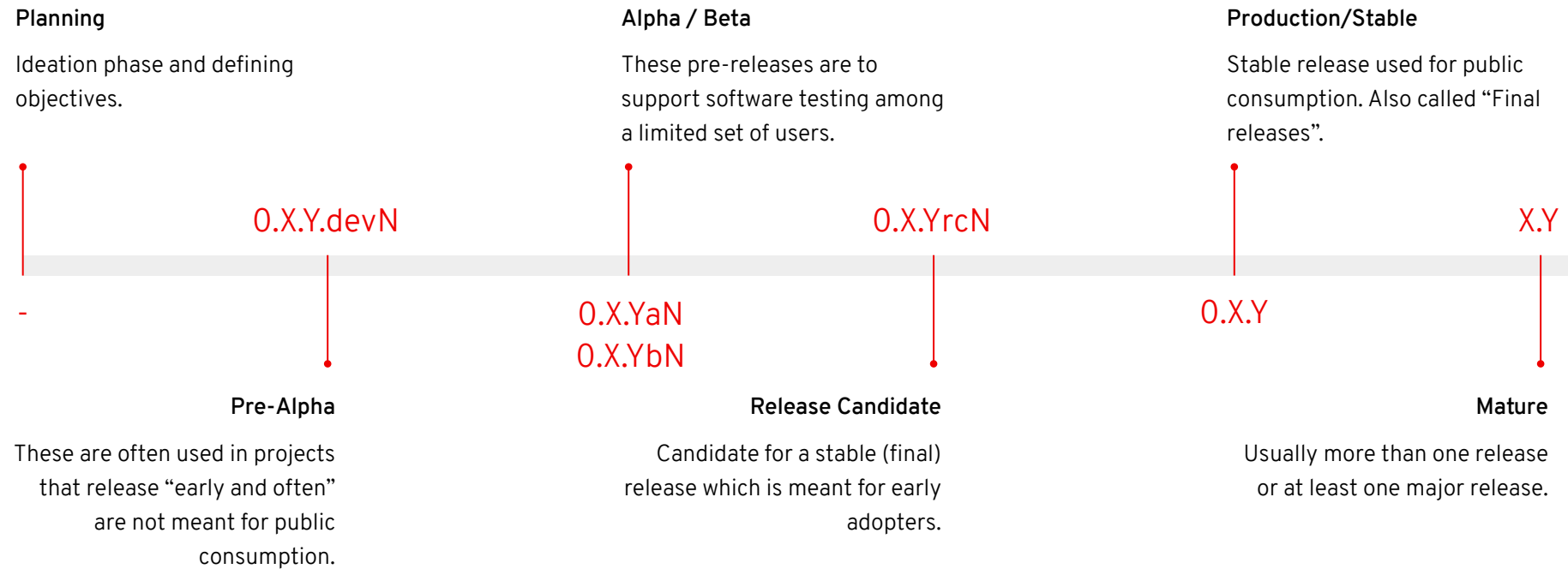
Semantic versioning

The **version scheme** is used both to describe the distribution version provided by a particular distribution archive, as well as to place constraints on the version of dependencies needed in order to build or run the software.

The canonical public version identifiers **MUST** comply with the following scheme:



Development status (Software release lifecycle)



Planning

Define objectives and the target audience

You gotta know WHAT the project is for and WHO the project is for. The objectives should be well defined and understandable.

Do the research

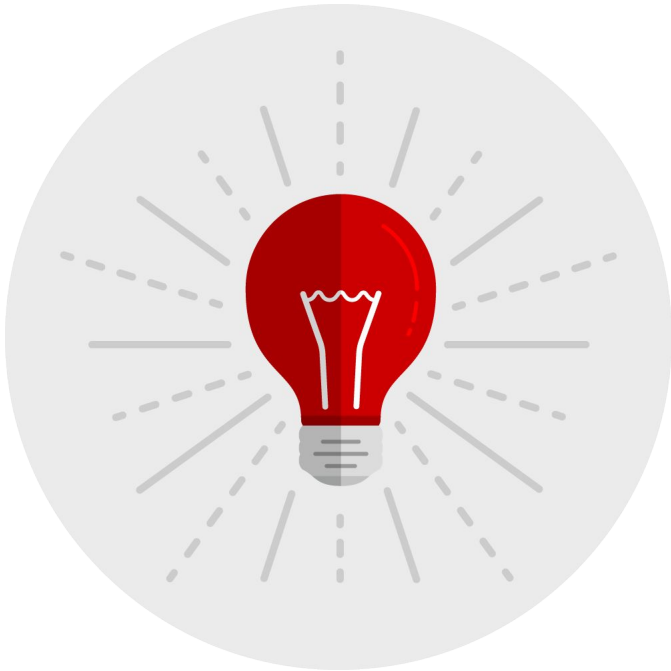
Do NOT reinvent the wheel!
If there is an existing project which is similar, see if you can use that one or contribute.

[if not defined] Determine the tools and delivery strategy

These can be publication tools, installation tools and other automation tools used for the development and delivery.

Come up with a POC

Prove the concept. This accounts for a feasibility study as well.



Pre-alpha refers to all activities performed during the software project before formal testing

These activities can include requirements analysis, software design, software development, and unit testing.

Software design

From a concept, through the architecture and implementation details.

Software development

Includes programming, feature implementation, feature enhancements, bug fixes or maintenance (i.e. updates and migrations, etc...)

Unit & Integration testing

Check whether the individual units of source codes function as expected based on a set of determined rules and whether they fit together and function together.



Pre-releases refer to a set of version identifiers which denote a preparation for the final release and are meant for early adopters

Among the pre-releases we include the alpha/beta releases and release candidates

Alpha

The first phase of software testing before releasing it to customers / users. In proprietary software, it is not common for a package in alpha release to be generally available. Alpha usually ends with a feature freeze.

Beta

The software is expected to have bugs which do not directly affect its functionality. The main purpose is to reduce impact on customers / users or to demonstrate and preview a product. A commercial *betaware* is usually available to limited set of users outside of the organization (**closed beta**) or publicly (**open beta**).

Release Candidates

A beta version with the potential to become the final product ready to be released. Minor fixes to fix certain defects are expected but NO new features or API changes should be made.



Release (stable, or final release) indicates that the software is stable, tested and ready to be used

Good to go.

General Availability (GA)

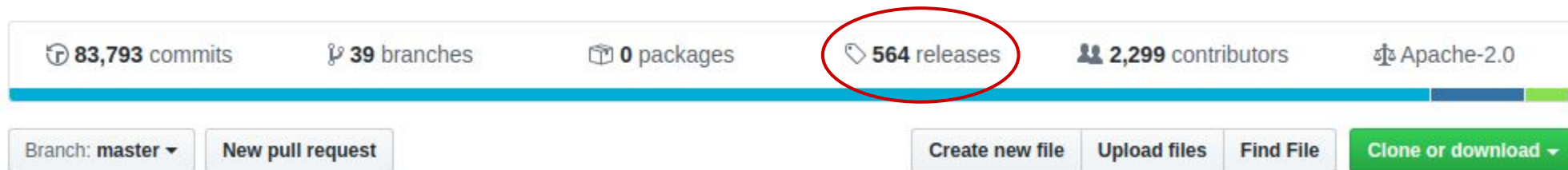
Used mostly for commercial products, but occasionally can be seen in the Open Source world as well. The GA means that the software is available for purchase.

Support

A release should be supported for a certain period of time and in further releases, there should be guarantee of certain backwards compatibility (this is not a rule, but is greatly appreciated).

So, when to contribute and when to file an issue?
When should I NOT use the project yet?

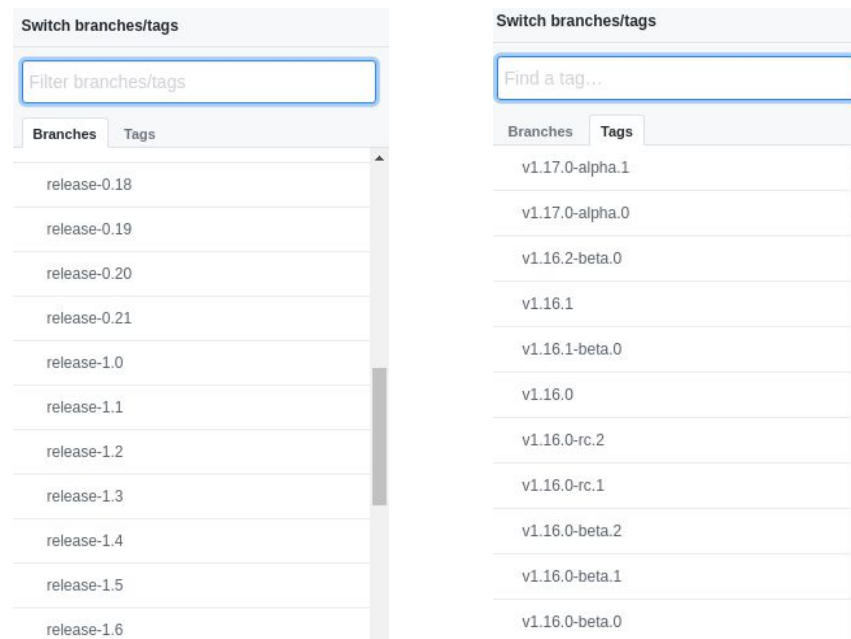
If a project is well-maintained, it is easy to spot the development status at the first glance.



The screenshot shows the header of a GitHub repository. The repository name is not explicitly shown, but the metrics are: 83,793 commits, 39 branches, 0 packages, 564 releases (circled in red), 2,299 contributors, and Apache-2.0 license. Below the metrics bar, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find File', and 'Clone or download'.

So, when to contribute and when to file an issue? When should I NOT use the project yet?

If a project is well-maintained, it is easy to spot the development status at the first glance.



So, when to contribute and when to file an issue? When should I NOT use the project yet?

If a project is well-maintained, it is easy to spot the development status at the first glance.

```
classifiers=[  
    "Development Status :: 2 - Pre-Alpha",  
    "Framework :: IPython",  
    "Framework :: Jupyter",  
    "License :: OSI Approved :: MIT License",  
    "Natural Language :: English",  
    "Operating System :: OS Independent",  
    "Programming Language :: JavaScript",  
    "Programming Language :: Python :: 3.6",  
    "Programming Language :: Python :: 3.7",  
    "Topic :: Utilities",  
],
```

Contribution Guidelines

Guidelines communicate how people should contribute to you project.

Contribution guidelines are a set of recommended practices, or sometimes even required ones, established by a maintainer for the contributors to be followed.

Before contributing to an open source project, make sure to check its contribution guidelines!

Usually, they can be found in a file called CONTRIBUTION.md^[0] or, in case of GitHub, they might be integrated to PRs and Issues directly^[1]



Verification for both contributors and developers

For both contributors and developers, the guidelines help them verify that they're submitting well-formed pull requests and opening useful issues.



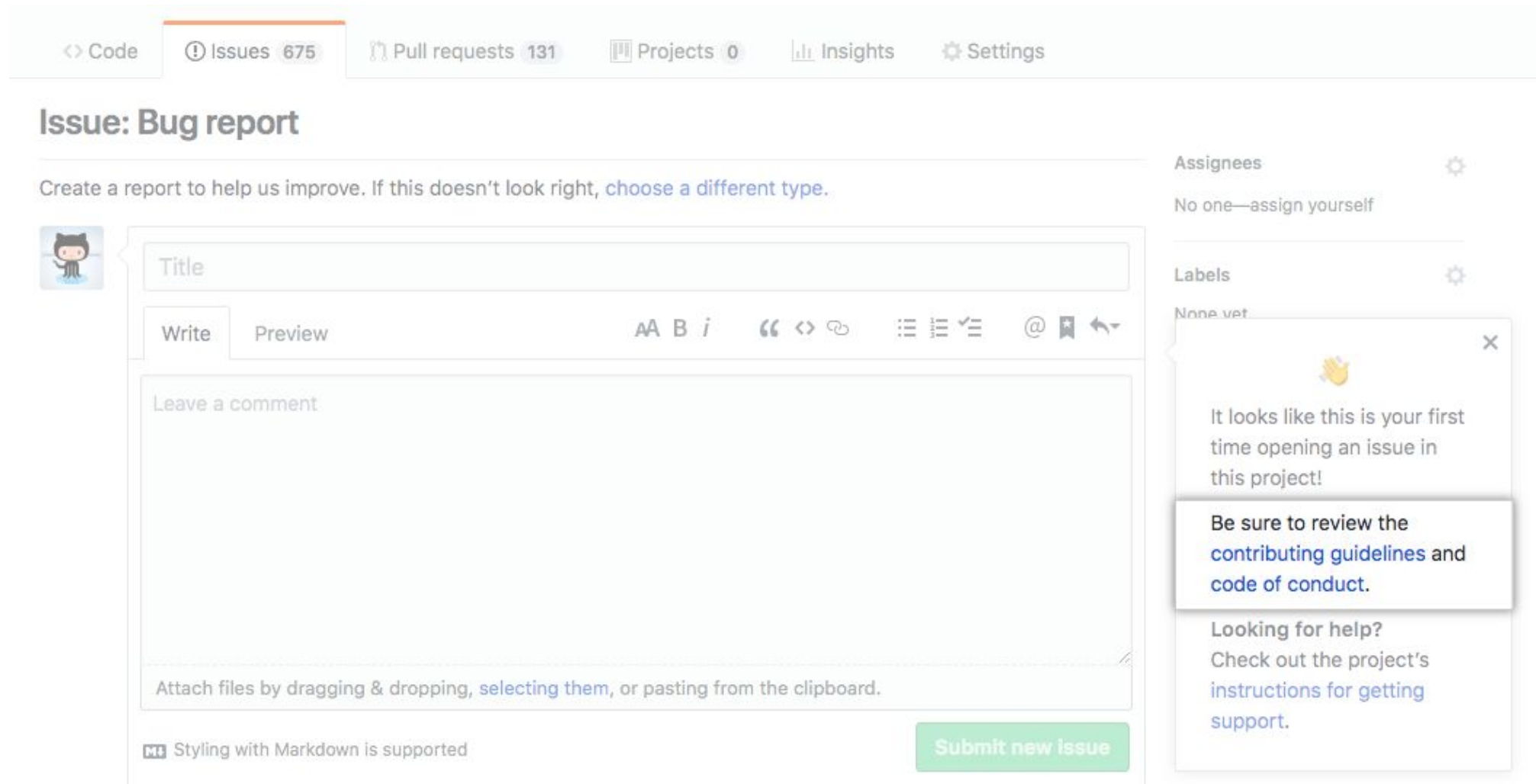
Getting started for contributors

Contributors might struggle to navigate in the project or they may not know where to start contributing, what should the PR or issue look like.



Prevent confusion and save time

For both owners and contributors, contribution guidelines save time and hassle caused by improperly created pull requests or issues that have to be rejected and re-submitted.



The screenshot shows the GitHub interface for creating a new issue. At the top, there are navigation tabs: Code, Issues (675), Pull requests (131), Projects (0), Insights, and Settings. The main heading is "Issue: Bug report". Below this, a message says "Create a report to help us improve. If this doesn't look right, choose a different type." The issue creation form includes a "Title" field, a rich text editor with "Write" and "Preview" tabs, and a "Submit new issue" button. A sidebar on the right shows "Assignees" (No one—assign yourself) and "Labels" (None yet). A tooltip is visible over the "Labels" section, containing a hand icon and the text: "It looks like this is your first time opening an issue in this project! Be sure to review the contributing guidelines and code of conduct. Looking for help? Check out the project's instructions for getting support."

As for the WHAT to contribute ... It doesn't have to be code

“““

Most people don't know that I actually don't do any real work on the CocoaPods tool itself. My time on the project is mostly spent doing things like documentation and working on branding.

@orta

“““

I first reached out to the Python development team (aka python-dev) when I emailed the mailing list on June 17, 2002 about accepting my patch. I quickly caught the open source bug, and decided to start curating email digests for the group. They gave me a great excuse to ask for clarifications about a topic, but more critically I was able to notice when someone pointed out something that needed fixing.

@brettcannon

“““

Okay, I read the contribution guidelines.
What now?
How do I proceed?
What are the best practices?



Ready-to-contribute pessimist Jerry

It all starts with a FORK ...

FORK the repository

A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Create a branch for the specific purpose

``git checkout -b fix-readme-typo``

Iterate on the issue

Not all fixes are so simple that they can fit into one commit.

Pull Request



Can you spot an ISSUE?

FORK the repository

A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Create a branch for the specific purpose

``git checkout -b fix-readme-typo``

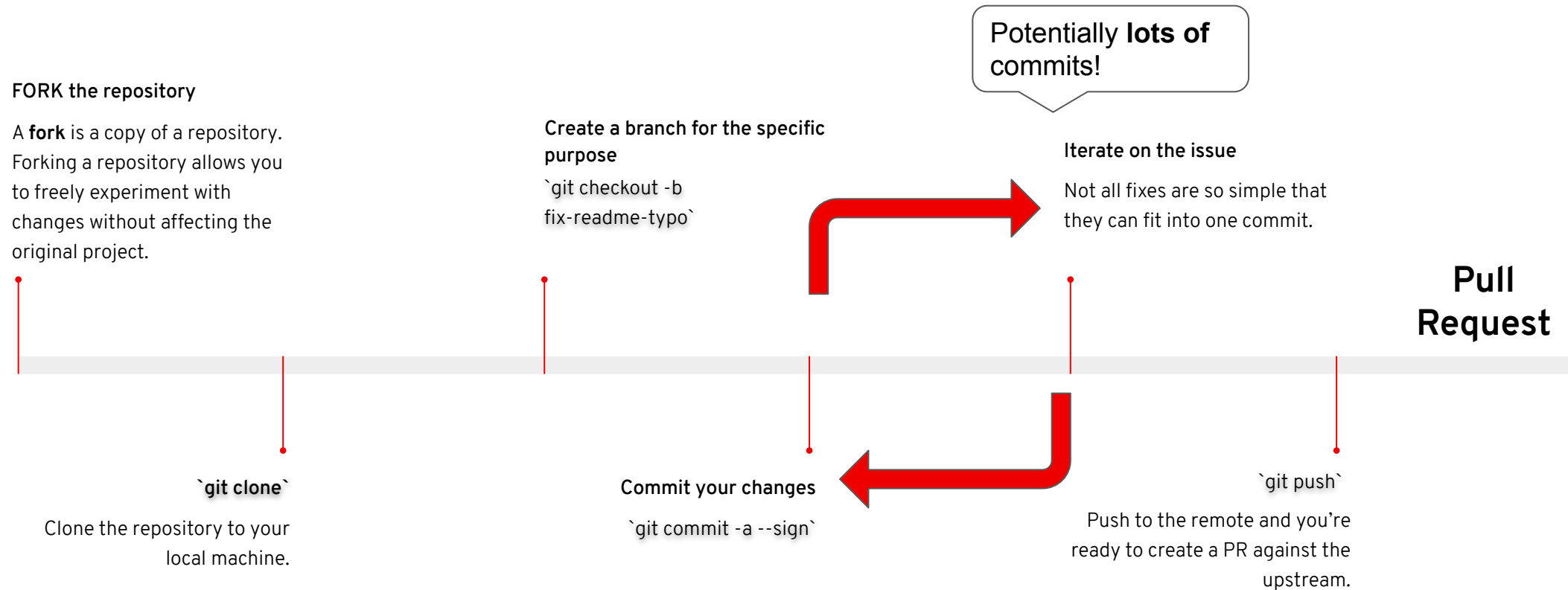
Iterate on the issue

Not all fixes are so simple that they can fit into one commit.

Pull Request



Can you spot an ISSUE?



Let's squash it!

FORK the repository

A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Create a branch for the specific purpose

```
`git checkout -b  
fix-readme-typo`
```

Iterate on the issue

Not all fixes are so simple that they can fit into one commit.

Pull Request



Can you spot an ISSUE?

FORK the repository

A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Create a branch for the specific purpose

```
`git checkout -b  
fix-readme-typo`
```

Iterate on the issue

Not all fixes are so simple that they can fit into one commit.

Pull Request



Pull, squash and push ... sounds weird, but does wonders!

FORK the repository

A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

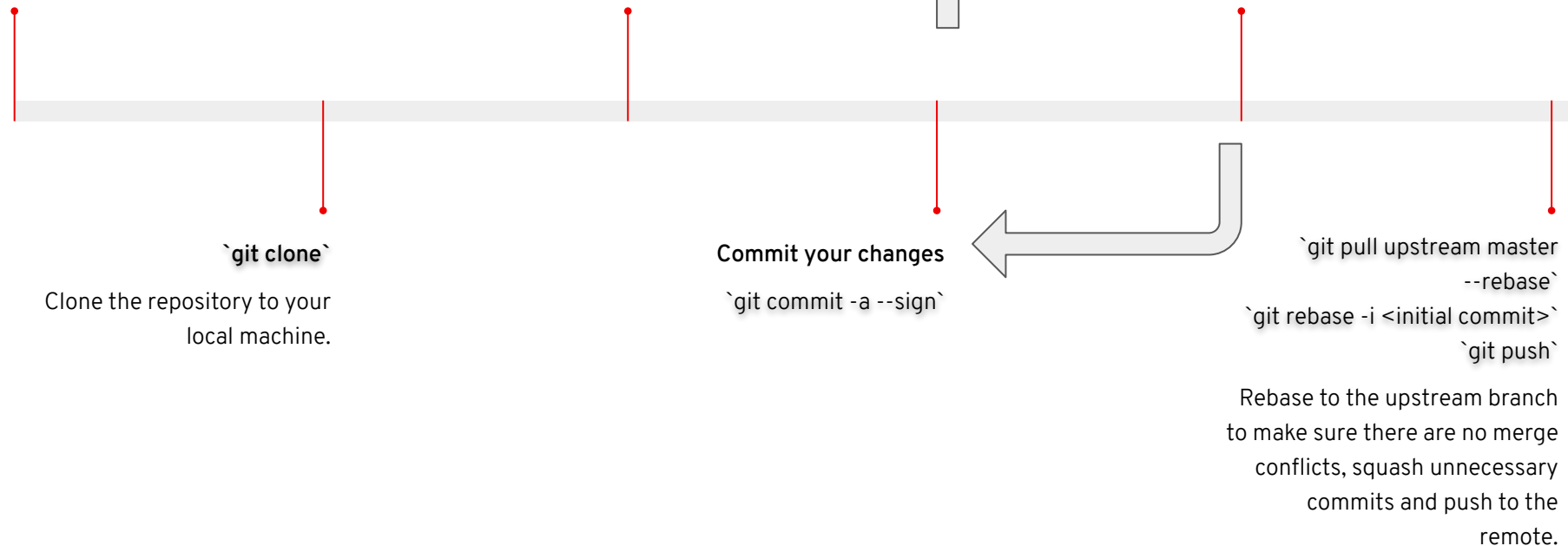
Create a branch for the specific purpose

```
`git checkout -b  
fix-readme-typo`
```

Iterate on the issue

Not all fixes are so simple that they can fit into one commit.

Pull Request



``git clone``

Clone the repository to your local machine.

Commit your changes

``git commit -a --sign``

``git pull upstream master
--rebase`
`git rebase -i <initial commit>`
`git push``

Rebase to the upstream branch to make sure there are no merge conflicts, squash unnecessary commits and push to the remote.



There are other useful practices to follow when contributing to the upstream



Document WHAT and WHY

Documenting why the changes have been made and what lead to the decisions you'd made will save the reviewer's time and will increase the chances of your PR being merged. Use provided doc generators, if possible.



Follow the code style

When contributing code, it is good practice to adapt to the project code style (especially for languages with fluid code styles, like JS). Use provided formatters, if possible.



Run tests before submitting the PR and write new ones when introducing new features.

Be extremely careful if changing project dependencies (see further)

Software dependencies

Dependencies are the hell for maintainers, a blessing for developers and the heaven for attackers.

A *dependency* is additional code that you want to call from your program. Adding a dependency avoids repeating work already done: designing, writing, testing, debugging, and maintaining a specific unit of code

Kinds of dependencies

Direct dependencies

Libraries that your code depends upon. These require some effort to control but comparing to the others they are sort of manageable.

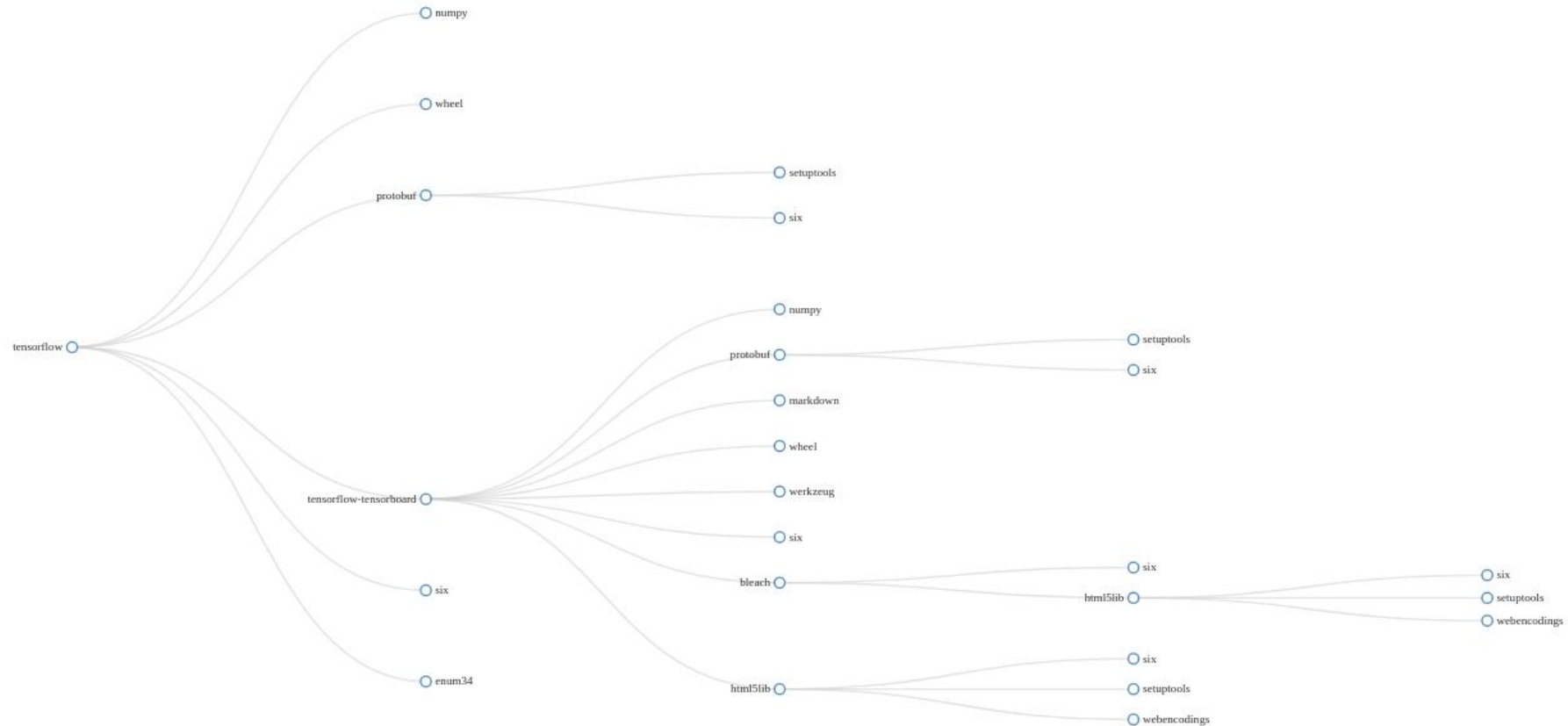
Transitive dependencies

Dependencies of the dependencies.
Usually quite hard to control.

Third party dependencies

A special kind. These are the dependencies that you don't own and that are not part of your organization. Especially hard to control.

Transitive dependencies



What could go wrong?

A package is code you **download** from the internet. Adding a package as a **dependency** outsources the work of developing that code—designing, writing, testing, debugging, and maintaining—to someone else on the internet, someone you often don't know. By using that code, **you are exposing your own program to all the failures and flaws in the dependency.**

Your program's execution now **literally depends on code downloaded from this stranger** on the internet.

What could go wrong? You name it ...



Security vulnerability (CVE)



Version conflict



Missing/Removed dependency



API Changes



License conflict



Broken third-party dependency

It sounds unsafe ...

And it *is*... but it is also necessary to keep the wheel of Open Source spinning!

A note about the security vulnerabilities

What is a "Vulnerability?"

An information security "vulnerability" is a mistake in software that can be directly used by a hacker to gain access to a system or network.

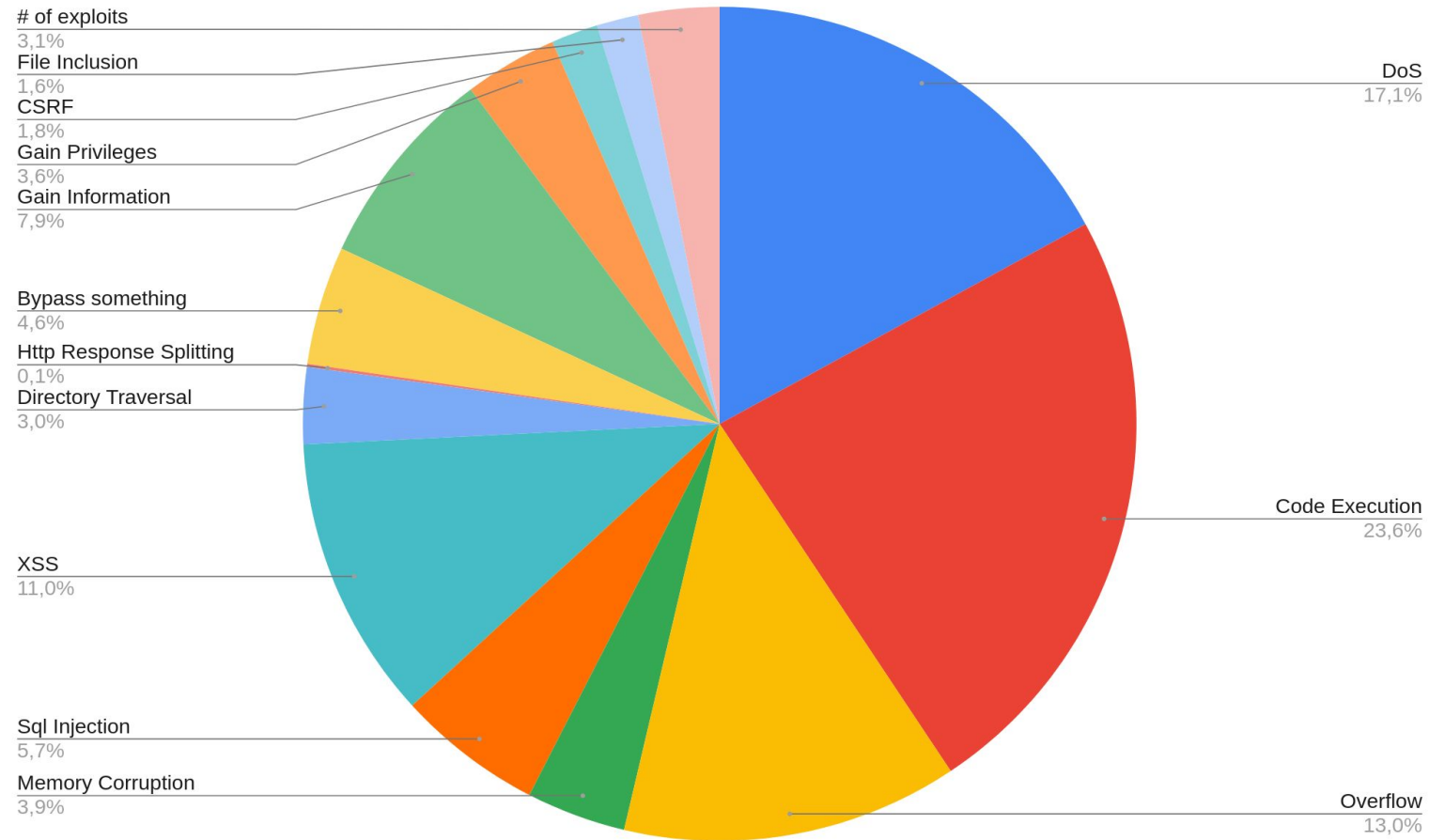
What is an "Exposure?"

An information security exposure is a mistake in software that allows access to information or capabilities that can be used by a hacker as a stepping-stone into a system or network.

What is CVE?

CVE is a list of information security vulnerabilities and exposures that aims to provide common names for publicly known problems. The goal of CVE is to make it easier to share data across separate vulnerability capabilities (tools, repositories, and services) with this "common enumeration." Please visit <http://cve.mitre.org/about/faqs.html> for more information

Common vulnerabilities according to the NVD



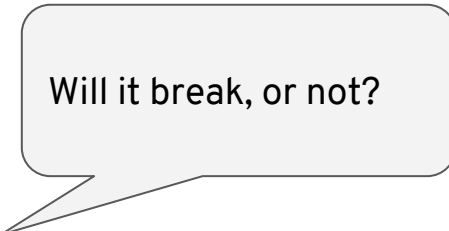
Beware the “dependency hell”

Especially when working with complex systems which have a lot of dependencies, it might be *incredibly difficult* to find the “right” **combination of versions** which are actually compatible together.

Sometimes, we might actually reach sort of a “**deadlock**” state if one dependency requires a version of another which is in fact not compatible with the rest of the project, i.e.:

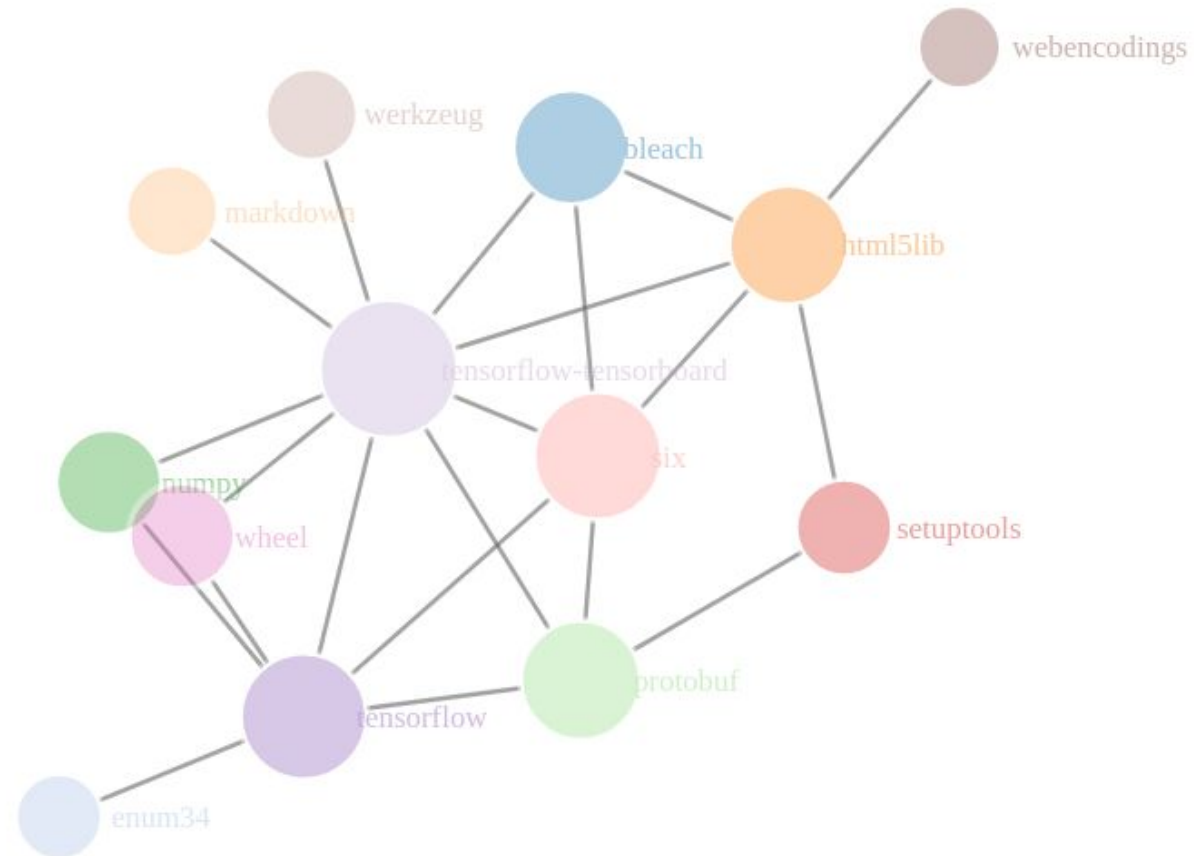
A requires D_a && D_a requires $X == 1.13$

A requires D_b && D_b requires $X == 1.13.5$



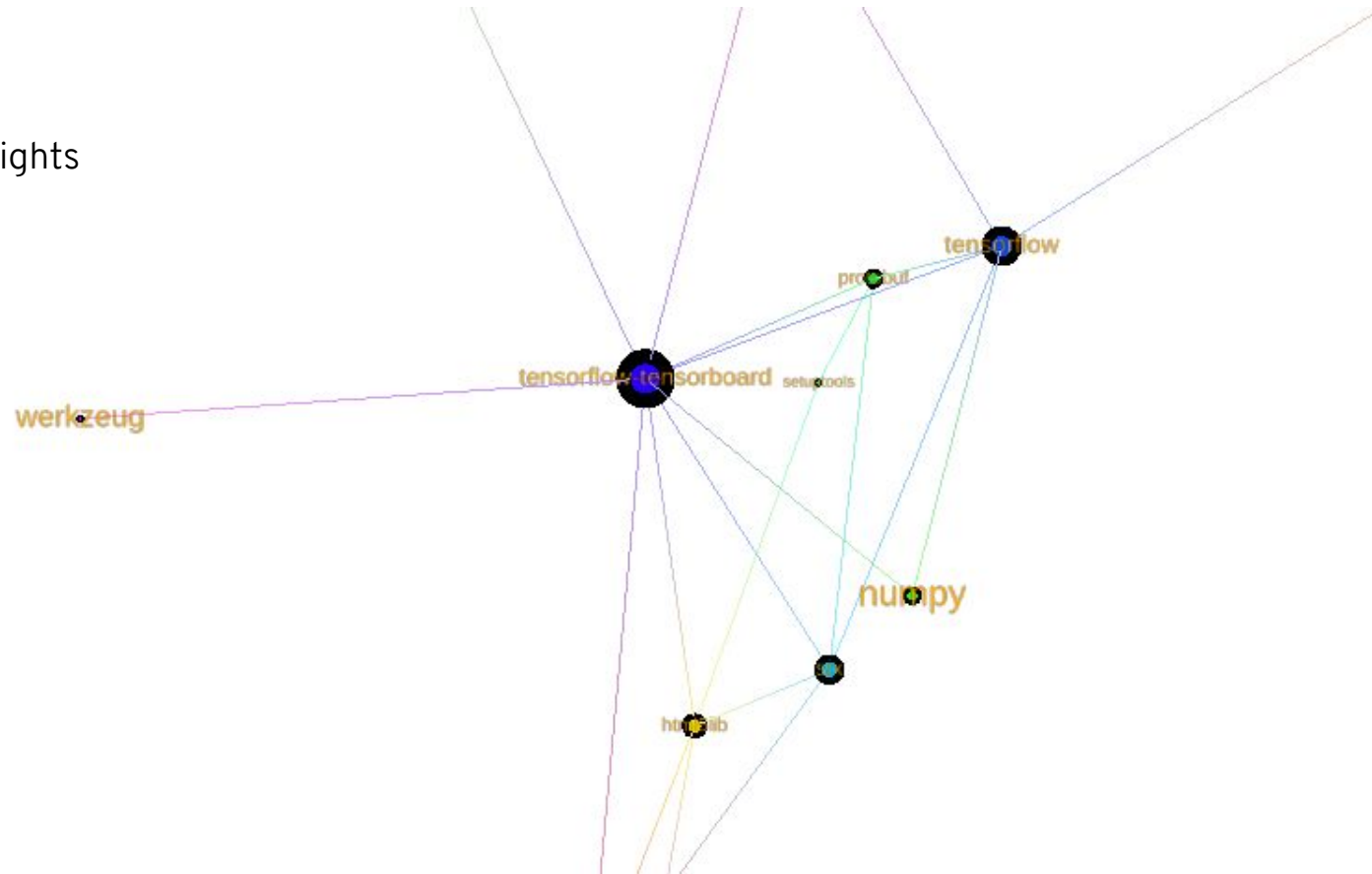
Will it break, or not?

It might become tedious ...



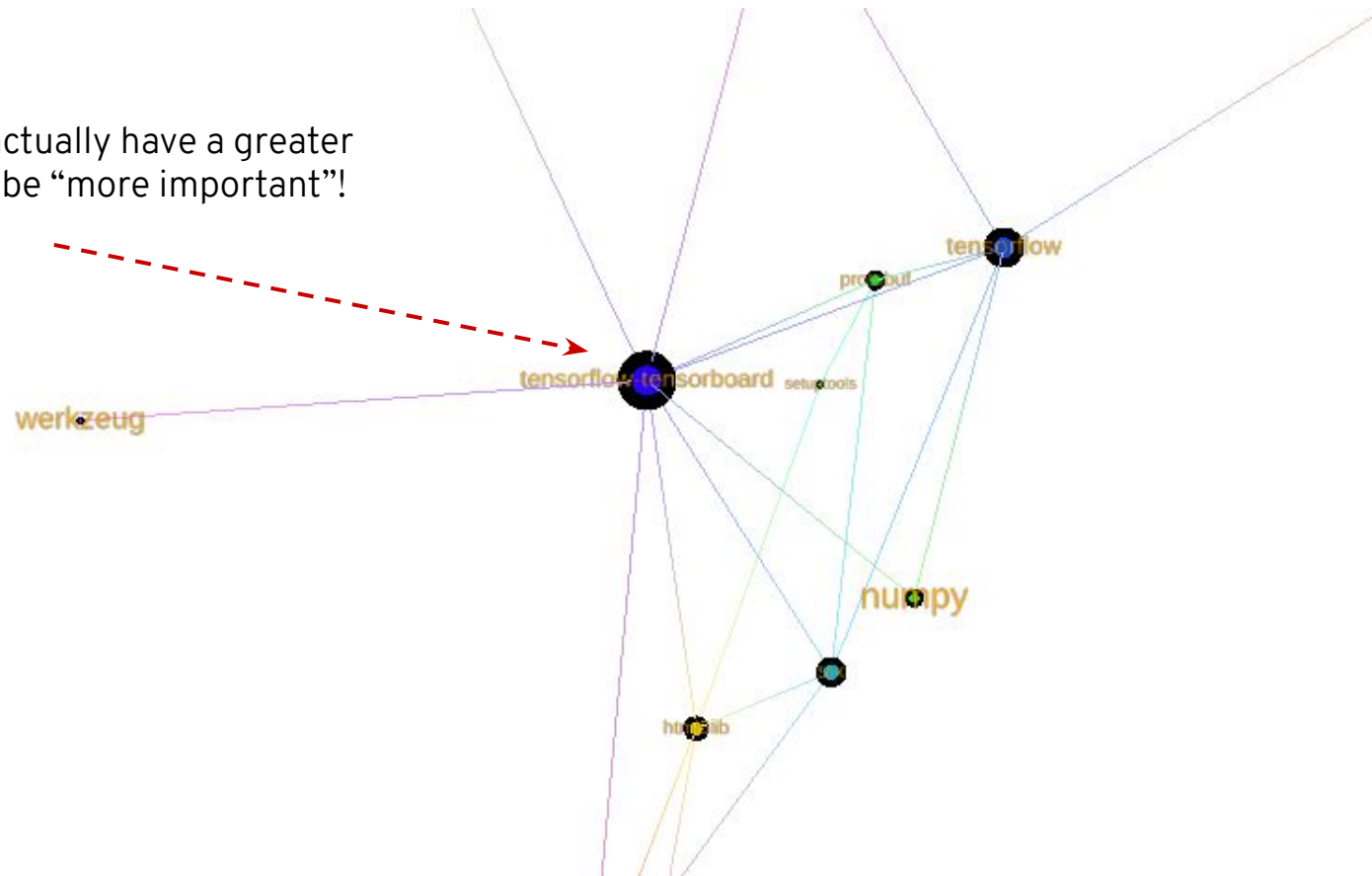
Can you guess the issue?

HINT: centrality, weights



Can you guess the issue?

A dependency can actually have a greater **centrality** and thus be “more important”!



So... what can we do?



Good practices when managing dependencies



Consider the value of adding the dependency

If introducing the dependency means a few lines of code that you're spared of, do NOT introduce the dependency at all. It is not worth it.



Choose a compatible and secure version

Take the time and investigate. Choose a version which is CVE free and is compatible with the rest of the application.



Consider the impact of the dependency

Consider how important the dependency is to your application and treat the dependency accordingly.



Keep your dependencies up to date

Update the dependencies and keep the code you own up to date with them. Do not rely on the pinned down version.



Unit TESTS & integration TESTS!

Write unit tests and integration tests especially for functions using a code that you don't own!



Regularly watch for CVEs and consult the NVD

Do NOT expose your application. GitHub and specialized software exist to inform you about potential security risks of your application.

And don't ever forget ...



TO MAKE SURE THAT THE **LICENSES** ARE
COMPATIBLE!

The compatibility is sometimes tricky ...

		I want to release a project under:					
		GPLv2 only	GPLv2 or later	GPLv3 or later	LGPLv2.1 only	LGPLv2.1 or later	LGPLv3 or later
I want to copy code under:	GPLv2 only	OK	OK [2]	NO	OK: Convey project under GPLv2 only [7]	OK: Convey project under GPLv2 only [7][2]	NO
	GPLv2 or later	OK [1]	OK	OK	OK: Convey project under GPLv2 or later [7]	OK: Convey project under GPLv2 or later [7]	OK: Convey project under GPLv3 [8]
	GPLv3	NO	OK: Convey project under GPLv3 [3]	OK	OK: Convey project under GPLv3 [7]	OK: Convey project under GPLv3 [7]	OK: Convey project under GPLv3 [8]
	LGPLv2.1 only	OK: Convey code under GPLv2 [7]	OK: Convey code under GPLv2 or later [7]	OK: Convey code under GPLv3 [7]	OK	OK [6]	OK: Convey code under GPLv3 [7][8]
	LGPLv2.1 or later	OK: Convey code under GPLv2 [7][11]	OK: Convey code under GPLv2 or later [7]	OK: Convey code under GPLv3 [7]	OK [5]	OK	OK
	LGPLv3	NO	OK: Convey project and code under GPLv3 [8][3]	OK: Convey code under GPLv3 [8]	OK: Convey project and code under GPLv3 [7][8]	OK: Convey project under LGPLv3 [4]	OK
I want to use a library under:	GPLv2 only	OK	OK [2]	NO	OK: Convey project under GPLv2 only [7]	OK: Convey project under GPLv2 only [7][2]	NO
	GPLv2 or later	OK [1]	OK	OK	OK: Convey project under GPLv2 or later [7]	OK: Convey project under GPLv2 or later [7]	OK: Convey project under GPLv3 [8]
	GPLv3	NO	OK: Convey project under GPLv3 [3]	OK	OK: Convey project under GPLv3 [7]	OK: Convey project under GPLv3 [7]	OK: Convey project under GPLv3 [8]
	LGPLv2.1 only	OK	OK	OK	OK	OK	OK
	LGPLv2.1 or later	OK	OK	OK	OK	OK	OK
	LGPLv3	NO	OK: Convey project under GPLv3 [9]	OK	OK	OK	OK

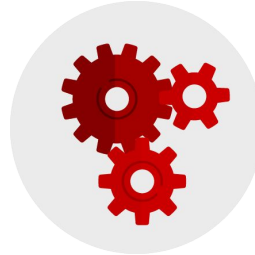
Continuous { Integration, Delivery, Deployment }

Introduction to CI/CD

CI/CD are the acronyms that are often mentioned when people talk about modern development practices.^[0]

CI/CD is a set of practices which have a significant impact to the way new releases are delivered and maintained.

These are the three main practices to be familiar with.



Continuous Integration

Change validation by creating a build and running automated tests against the build. By doing so, you avoid the integration hell that usually happens when people wait for release day to merge their changes into the release branch.



Continuous Delivery

An extension of continuous integration to make sure that you can release new changes to your customers quickly in a sustainable way. This means that on top of having automated your testing, you also have automated your release process



Continuous Deployment

Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production

Q&A



