DevOps

Objectives

Software engineers need tools, infrastructure, and good practices when designing, writing, deploying and maintaining software. The term DevOps comes from a contraction of software development (Dev) and IT operations (Ops), in recognition of the fact that the two are intertwined. DevOps aims to shorten the software development life cycle and to provide continuous delivery of functionality while preserving high software quality. In this context, we will discuss tools like **version control systems**, which allow developers to track code and code changes, ensuring that all members of a team can work on the same codebase without unnecessary conflicts. **Build systems** allow developers to automatically compile and run their code, removing the risk of human error in complex manual processes. **Continuous integration systems** allow developers to see the effect of their changes as they go, providing fast feedback that is essential to reduce defects.

In this module, you will learn:

- How to keep track of code and manage changes
- How to automatically compile, run, and test code
- How to detect defects as soon as, or even before, they are introduced in a codebase

Software Product vs. Code

The future belongs to software, and the reason you are studying software engineering is because you want to be part of that future. It doesn’t matter if you work in the car industry, in healthcare, in finance, or in biology, you will have to write software. This is the new literacy and, today, anyone who wants to solve real problems must write software.

So far in your studies you've learned how to write programs, and that is the basis upon which the present course builds. As you will see, there is an important difference between programs and software, between the code you write and the useful artefact that people get to use. Hospitals are employing computer software to diagnose diseases. We use Zoom and Skype to do our work, online mail and calendaring clients to keep track of our life. Washing machines, medical devices, and the stock market run code. A modern self-driving car is full of computers and code. These are all products, not programs.

Software is not a program. When you want to deliver a software product to human users, you need to assemble programs into a system whose pieces all work well together.

Software that works well has five key attributes:

- *Reliability* means that the software does correctly what you ask it to do. A reliable email system will accept your message and deliver it to its recipient, as promised…
- *Safety* means that, even when software happens to be unreliable, the failures it does admit do not have catastrophic consequences. For example, a safe cellphone is one that does not stop working when you need to call an ambulance.
- *Security* means that the software system protects your information from theft, unauthorized access, or tampering. For example, a self-driving car would not allow remwote hackers to gain access to your vehicle and crash it while you're in it.

- *Performance* means that the software meets users' latency and throughput expectations. For example, when you do a videoconference, you expect video frame delivery to be smooth and to not have hiccups in the audio.
- *Manageability* means that configuring the software and keeping it operational can be done flexibly, quickly, and without high risk for error.

There exist other desirable properties too for software (e.g., the software running on your smartphone should not consume too much energy), but in this course we will focus on the five attributes above.

In this chapter we will take a first look at the road a program must take to become software.

Developer Tools

First, let's do a quick tour of what you are likely to find in the toolbox of a modern software engineer. Those who are fluent in using these tools become much more efficient at writing high quality software.

The building blocks of any programmer's toolbox are the preprocessor, compiler, linker, and interpreter.

Generally speaking, a **preprocessor** is simply a program processing its input to produce some output data used as input for another program. An example is the C preprocessor, which is used to perform macro expansion, conditional compilation and header files inclusion. Other relevant examples include modern web tools, such as Sass and Babel, that bridge the gap between newer language features and legacy code. A **compiler**, instead, translates source code into binary (or object) code, which can be directly executed by the machine after linking. Examples of compiled languages are C and C++. Finally, an **interpreter**, compared to a compiler, directly executes the instructions written in the source code or some intermediate representation (like Java bytecode), without previously converting them into a machine code program. Examples of interpreted languages are Perl and Python.

To improve their efficiency, software engineers like to automate the whole process of compiling and linking their programs. A **build system**, or build automation tool, automates the creation of executables, incorporating tasks such as compiling, linking and packaging the code. It can also be used to manage dependencies, run tests and deploy artifacts (more on this in the Continuous Integration section). Build systems make developers' lives easier by abstracting away most of the complexity, automating a variety of tasks that they would normally have to invoke manually (and in the right order) to complete the build process. Examples are Gradle, Maven, sbt and Make.

Developing reliable and bug-free software is a hard and complex task, and testing it (preferably using a framework) is the most widely used way to improve the chances that it behaves as expected. A **testing framework** is a set of practices and tools used for creating and designing tests. It is application-independent and can expand with the requirements of each application. As such, it helps teams organize their test suites and improve their efficiency, as they have to learn just one framework, which can then be reused across multiple products. Examples are JUnit, Espresso (which you will both use), Jest and pytest.

Tests are great, but sometimes they are not enough to pinpoint the source of a bug: in those cases, you need something more. A **debugger** is a tool used to test and debug programs. It is mainly used to run the target program under controlled conditions, track control flow and monitor changes in resources. Examples are GDB and the Android Debugger. GDB can, for instance, be used to easily discover the source of a segmentation fault in C/C++ code.

Having well-written and tested code is unfortunately not enough; performance is important too. To measure it, developers use a tool called a **profiler**, to analyze the performance of their applications. A profiler can either instrument a program's source code (or binaries) and track its resource usage (CPU, RAM, etc.) or it can build statistical, sampling-based summaries (called profiles), containing useful information about the frequency and duration of function calls, by periodically inspecting the call stack. It thus can be used to identify sections of code where the application is making inefficient use of resources. A good example is the Android Profiler.

Documenting your code is as important as writing and testing it. A **documentation tool** helps generate, in an automated fashion, software documentation from a set of source code files that have been suitably annotated. Popular examples are Javadoc, Doxygen or Sphinx, which use special comments embedded in the code to drive the generation of the documentation.

Is this it? Yes, if you're willing to risk losing your work because of a failure of your hardware or an error made by your teammates. If not, then you will want to use a **version control system** (VCS) that records changes to your source files over time. It is one of the most important and essential tools to work with, as it allows you to revert files to a previous state, compare versions over time and see who authored a change that introduced a problem. Moreover, it can also help restore files erroneously deleted. Examples of VCSs are Git, Subversion and Mercurial. Software of any magnitude also requires a **ticket system**, or an issue tracking system, which manages lists of issues, bugs and tasks. Examples include Jira as well as GitHub's system (a useful guide on it is available here).

Finally, if you work in a collaborative setting, you will need **continuous integration**. This is a development practice where multiple developers integrate code into a shared repository frequently. Each integration is then usually verified via automated builds and tests. The main benefits of this approach are early bug detection and the fact that individual contributors' code does not divert much. Combined with continuous delivery or continuous deployment (CI/CD), it forms the cornerstone of modern DevOps. Examples of continuous integration services are Travis CI, Jenkins and GitHub Actions.

When building software, we often find ourselves reusing functionality that many others have used before us. An **SDK**, or **Software Development Kit**, is a set of tools and libraries used to develop applications on specific platforms. An example is the Android SDK, bundled with Android Studio, which includes a debugger, libraries, an emulator, documentation, sample code and tutorials.

The aforementioned Android Studio is a good example of an IDE, or **Integrated Development Environment**, which is an application that puts at the developer's fingertips many useful tools, such as a source code editor, a build tool, debugger, profiler, a compiler, and version control. Other examples are IntelliJ IDEA, Eclipse and Visual Studio.

During your journey in this course, you will encounter the Android Emulator, a software system that emulates Android devices on your computer so that you can run and test Android apps on multiple devices and Android versions without the need to use a physical device. In general, an **emulator** is a software system designed to behave in the same way as some other system. Another example is a Gameboy emulator that allows you to play good old Gameboy games on modern Windows machines.

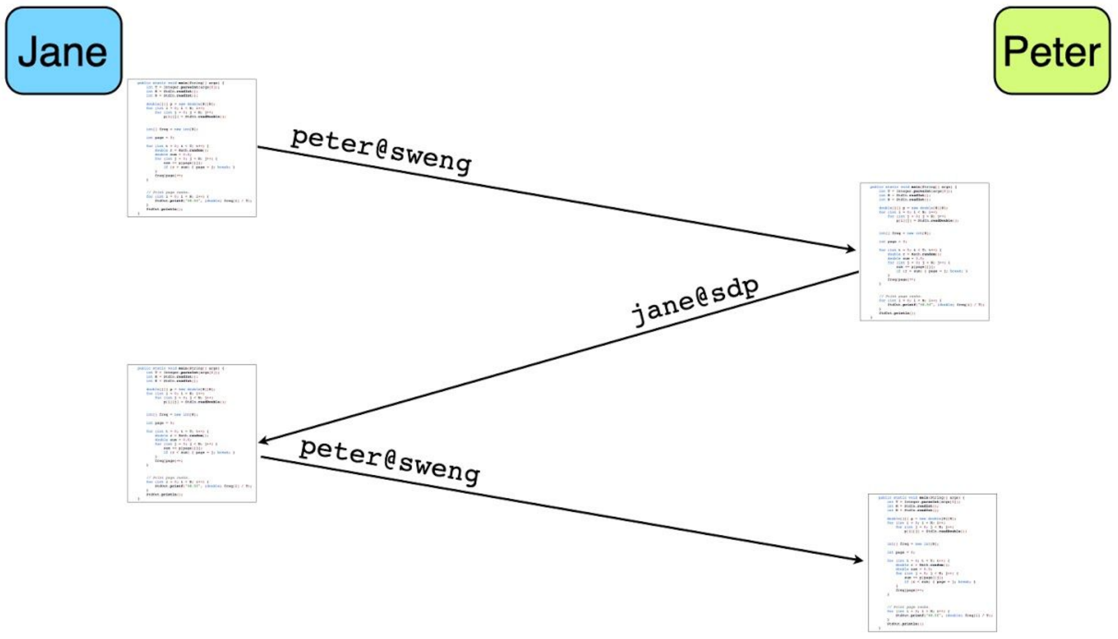
Managing Source Code

What is a version control system?

In this section we will see one of the most important tools that allow engineers to cross the gap from programs to software: source code management systems. When multiple engineers work on a codebase, it becomes important to have a systematic way of dealing with different versions of the code, and that is where a version control system, or VCS for short, comes in. Version control is one of the most basic forms of source code management.

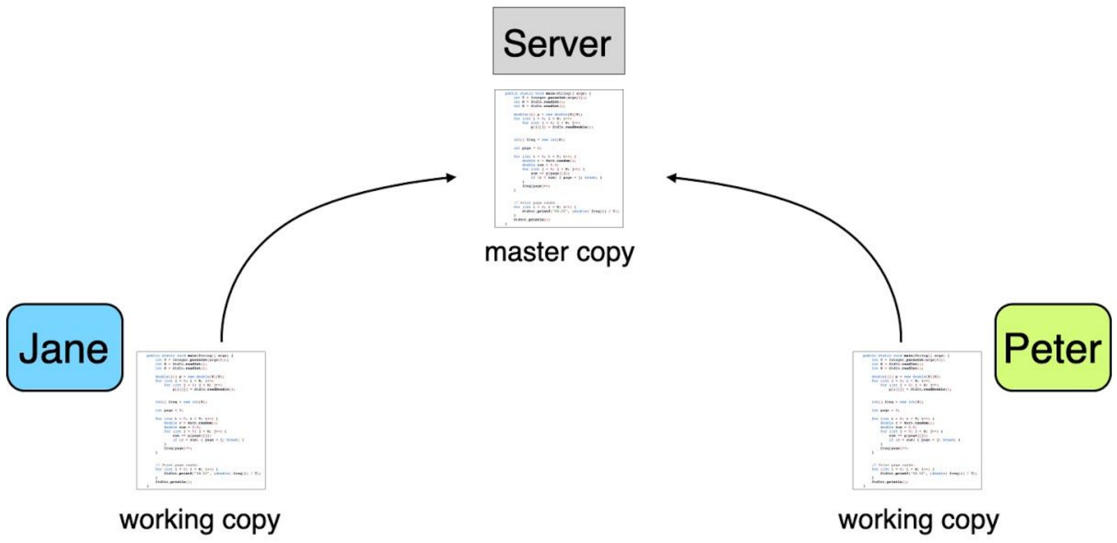
When you write code on your own, say you're writing a small smartphone app, you just add the features one after the other, compile, debug, etc. and when you're done, you upload it to the app market and you're done.

Now say that there are two developers who want to work together on a more serious app, call them Jane and Peter. They need to edit the same code. One way this could happen is for Jane to work on it, then ZIP up the source tree and email the code to Peter and say "ok, now you work on it", then when Peter is done with his part he emails it back to Jane and so forth. This essentially serializes access to the code.



A better alternative is to have Jane and Peter work on the project in parallel with each other and then "merge" their changes periodically. This is difficult and time-consuming to do over email, so they would probably need to get on the phone or a videoconference and discuss how to merge the changes.

Another possible model here is to have a "master" copy of the project in a central location, perhaps on a server somewhere in the cloud. Every time Jane or Peter have a set of changes that they made, they can go and merge these changes into that master copy.



The idea is that they first make changes to their own copy of the code, called the "working" copy, and when they have a stable set of changes, they can merge the working copy with the master copy. This way, Jane and Peter can work independently, making whatever changes they want, and they don't need to meet with each other and discuss, or synchronize over email.

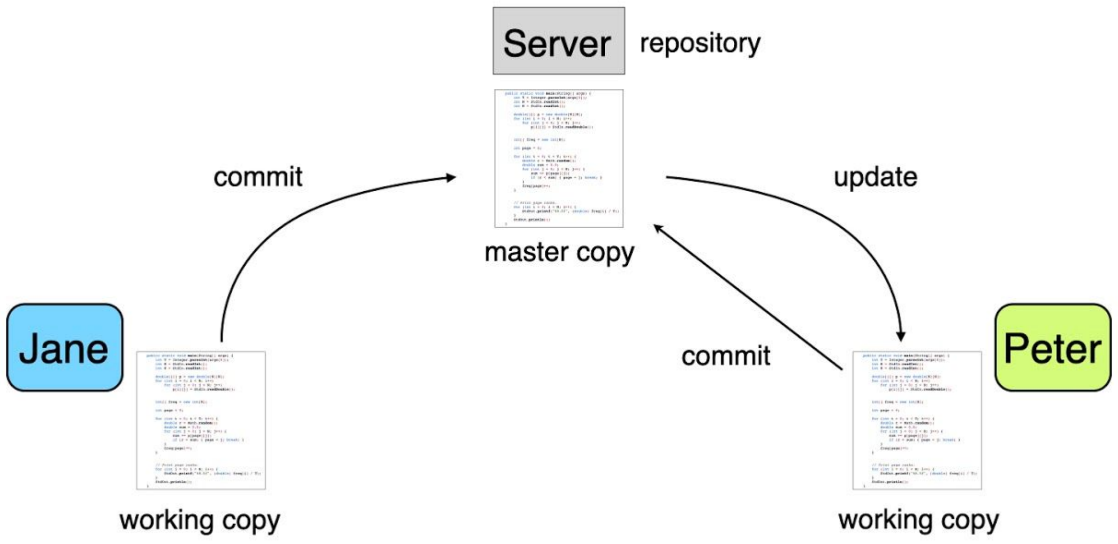
The approaches we've seen thus far might work well for two developers, but when you have a larger team, this does not scale. For example, the more developers work on a code base, the more frequently the master copy will change. This means that Jane might make a working copy, merge her changes, and then when she wants to replace the master copy with hers, it turns out the master copy has changed in the meantime. So she needs to repeat the process, and she might have to keep doing this forever. This is inefficient.

An alternative would be for Jane to lock the master copy while she is merging her local changes. But this means that other developers will not be able to advance, and they will have to wait for Jane. The more developers work on the same codebase, the more each one of them will have to wait. In other words, the overhead of coordinating between developers becomes high.

The first generation of version control systems (such as RCS, first released in 1982) worked approximately in this way, with a few tweaks as we'll describe below. To mediate collaborative development of code, these VCSs also automated to a large extent the storing, retrieval, logging, and merging of revisions made to the codebase.

In the classic model, your changes were made to your working copy, and they would be "committed" to the master copy once you were ready to merge your working copy into the master. Once committed to the master copy, all other developers who have access to the master copy would see your changes. They could then update their working copy correspondingly, make changes, and then commit their changes to the master copy.

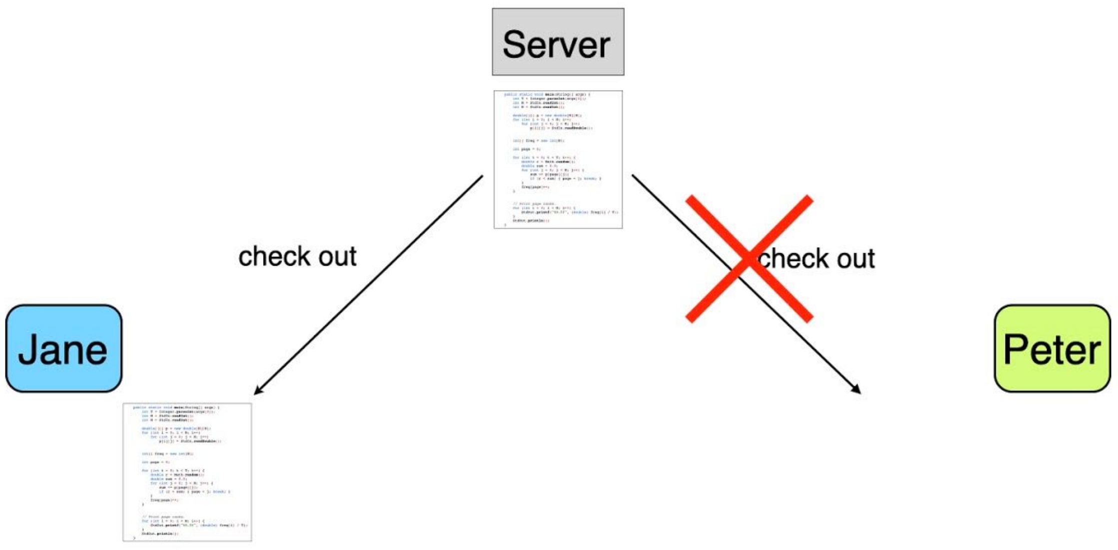
The place where the master copy resides is usually called a "repository".



As you might suspect, managing source code in this way is OK for small teams, but does not work at all for larger teams.

But what if Jane and Peter make changes to the same portion of code in their working copies? Clearly their changes are mutually exclusive, since they touch the same lines of code.

In first-generation VCSs, if a developer wanted to edit a file, that file had to be first "checked out" from the repository; this would prevent anyone else from checking it out. In other words, they only locked the files they edited instead of locking the entire master copy.



Once the developer checked the file back in, others could check it out and edit it. This is a bit like the model of emailing code back and forth, but at the granularity of individual files, and the system takes care of a number of the logistics. Since it is not the entire master copy that is locked, then multiple developers can work in parallel, as long as they don't touch the same files. Furthermore, as the project scales up and the developer team grows, it is quite likely that the codebase itself grows in terms of number of files, and so the additional developers get somewhat distributed across the various files.

However, there will always be some "hot" files that a lot of developers want to touch. So this mutual exclusion model can significantly hurt productivity in larger teams.

To address this challenge, 2nd generation of version control systems (like Subversion, Perforce, or Rational ClearCase) made it possible for Jane and Peter to independently edit the files.

Then Jane may commit her changes to the repository and update the master copy; when Peter tries to commit his, the system notices that Peter's changes overlap with the changes made by Jane since Peter last updated his working copy, and prevents the commit from happening and indicates a conflict.

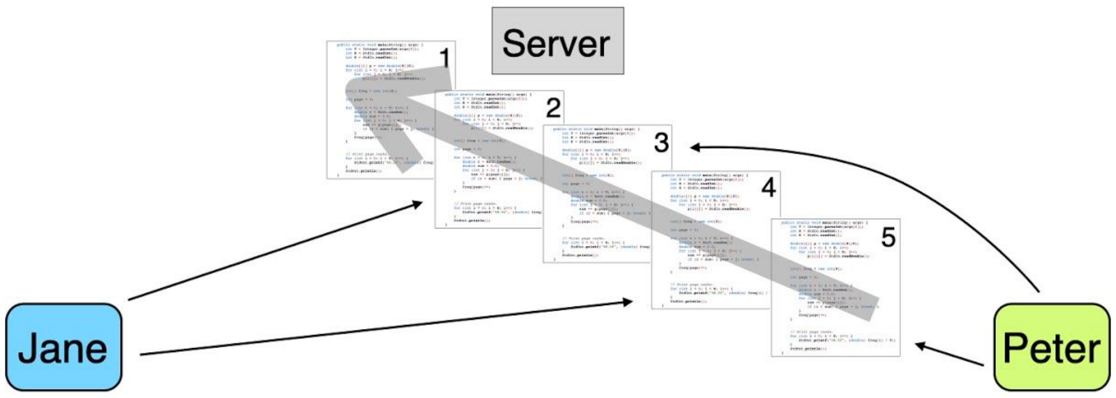
It is now up to Peter to edit the code to resolve this conflict: he can discard his own changes and keep Jane's, or he can overwrite Jane's, or modify the code in a way that accommodates both changes. Once Peter has fixed the conflict, he can commit to the master copy.

The VCS makes it harder for a developer to overlook or accidentally overwrite another developer's changes. Furthermore, the VCS can automatically highlight the portions of the code that are in conflict, while letting the non-conflicted portions go into the repository.

The other major benefit of such a VCS, and perhaps by far the most important one, is that the changes Jane or Peter commit to the repository do not overwrite the master copy, rather create a new version of it. In this way, the repository stores all versions of the code, and therefore can offer developers a clear timeline: the most recent code is the latest version, and this version is more recent than the version before it, and so on. The VCS then allows developers to "travel back in time" and retrieve any of the previous versions of the code.

For example, say you are working on your homework, and you've implemented 80% of the required features, and they all work fine. You commit that code to the repository. Then you continue working on the remaining 20% of the features, and you accidentally introduce a bug that breaks the code that worked fine before. What do you do? You can "travel back in time" by reverting to the version in which at least 80% of the features worked well, and redo your work more carefully.

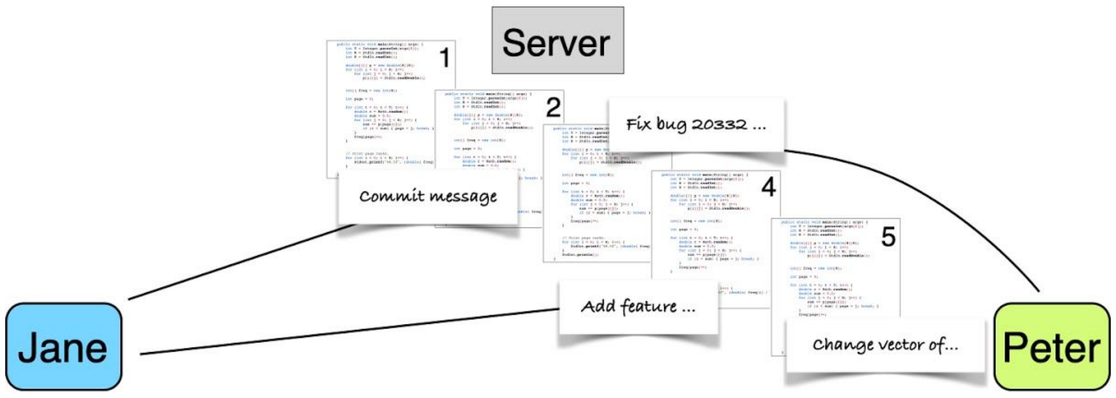
In other words, with a VCS you conceptually get the ability to undo every major change, all the way to the beginning of your project.



Commits, conflicts, and merges

As mentioned earlier, an atomic change to the code base is a commit. Whenever you commit a new version to the repository, a VCS will always allow you to include a commit message, which allows you to briefly describe what that change encompasses. This message captures the gist of what is special about that version. For example, it might say that the version you are committing fixes a particular bug, or implements a particular piece of functionality.

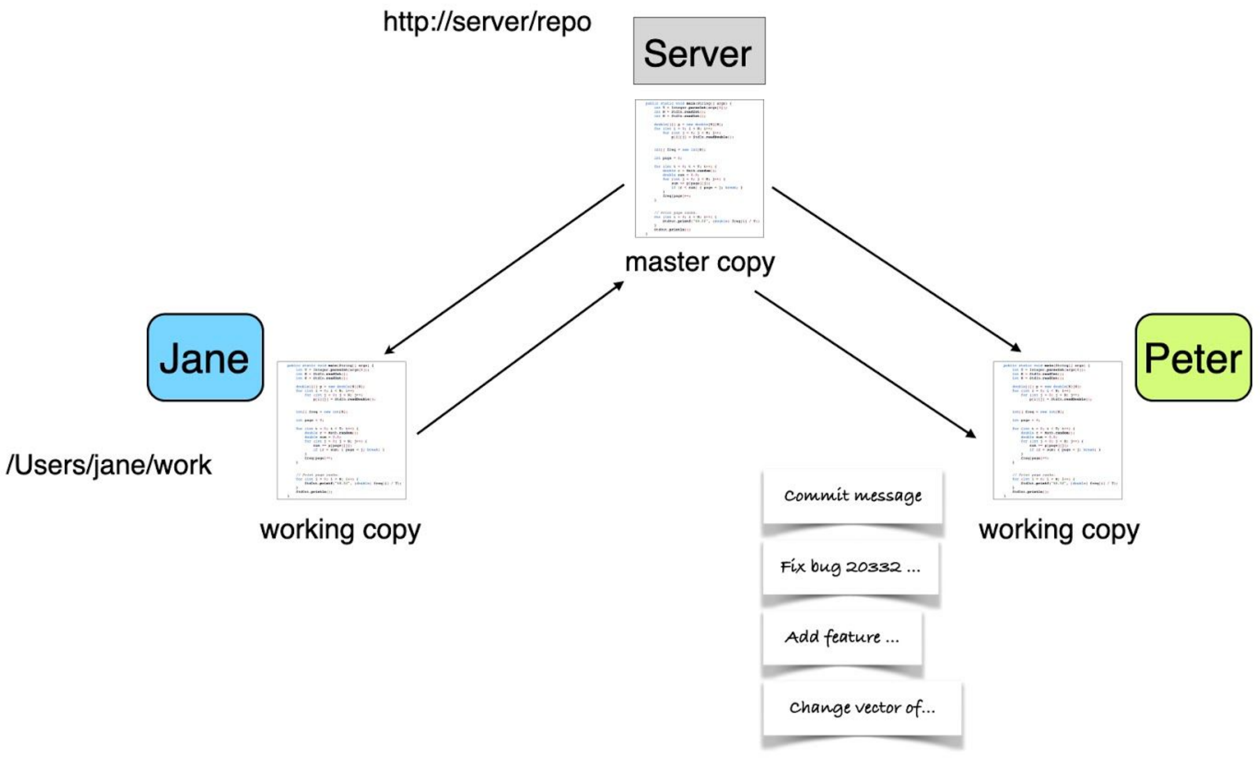
This way, if you want to travel back in time, it becomes easy to navigate by inspecting the messages and figuring out which version you are looking for. Additionally, other developers can read the commit messages and better understand what that version is about, even though they themselves did not write it.



This idea of establishing baselines and then making it easy to go back to previous versions is essential to a good VCS. If something goes wrong, a VCS allows you to determine what has changed and who changed it, so that you can quickly fix the problem

To recap, the typical workflow for a second-generation VCS revolves around a central repository that stores the master copy and all its versions, while Jane and Peter have their own working copies. The repository usually has some URL, because it sits somewhere on a server, whereas the local copy is usually in a folder on the developer's computer.

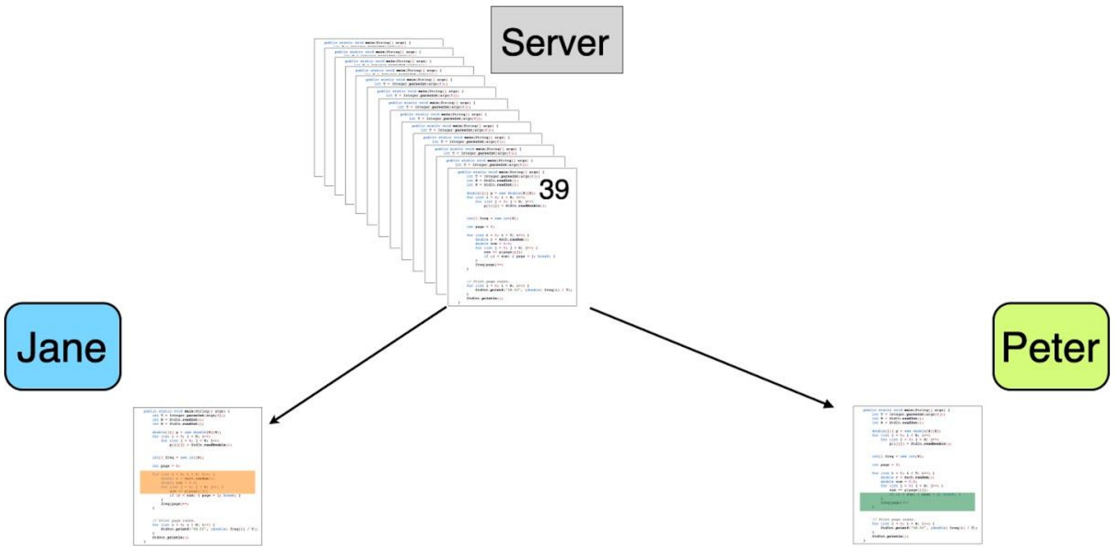
To obtain a working copy, Jane checks out the project code from the repository, and so does Peter. Subsequently, they can obtain updates incrementally, by updating their working copy and bringing it in sync with the master copy. Every time Peter does an update, he gets from the repository whatever changes Jane and all the other developers made. Peter can also inspect the log of commits, which includes a variety of information about each version, including the commit message, revision number, date of commit, the developer's name, etc.



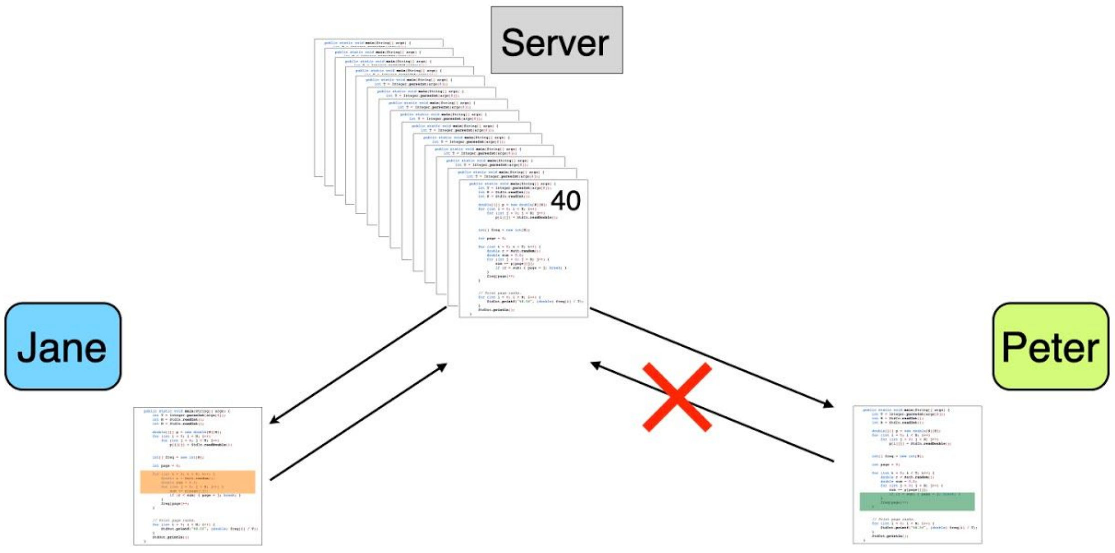
The most recent revision of the code is often called the HEAD. This is because it's the head of the timeline. Each previous version is typically a complete snapshot of the project, as opposed to a per-file version, in order to provide developers with a consistent view of the code base, as they travel back in time. What this means is that a version number characterizes all files in a project, together, so when you refer to revision 39, that refers to a complete version of your project.

This is also why all changes contained in a commit to the repository should have a single logical purpose, so that they're easier to find, understand, and operate on (e.g., if you want to undo some changes). For example, you may have fixed a bug in the code and also added a new piece of functionality. You may be tempted to bundle these unrelated changes into one single commit, but that is generally wrong, because say another developer later on realizes that your fix to the bug was incorrect, or not efficient, and wants to undo that fix (and that fix only) and repair it differently. By reverting to a specific version, the developer will end up undoing both the functionality put in that commit and the bug fix, and has no way to tease them apart. It is therefore better to commit the added functionality separately from the bug fix.

So how do changes get merged? Let's say Jane and Peter are each at revision 39 of the project, and they both want to modify function `foo()`.



Jane modifies the first 5 lines, then Peter modifies the last 3 lines, and then Jane sends her changes to the repository. When Peter tries to send his changes, the system tells him his working copy is out of date, so he needs to update his copy to revision 40.



When he does so, the system will figure out that the changes do not overlap, and will merge them into his local copy. At this point Peter reviews these changes, makes whatever modifications are needed, and then can send his changes (which are now applied on top of Jane's changes) to the repository.

Sometimes changes cannot be so easily merged, for example if Jane and Peter edited the same lines of code, in which case Peter would have to manually resolve this conflict. Depending on the tool you use, these diffs can be nicely highlighted in a way that makes it easy to see what differs in which file. One good practice is to identify the differences between a working copy of the code and the HEAD in the repository before updating, so that it is clear what will happen on an update.

If you find that code conflicts arise frequently, then this typically means that there is some deeper project management problem—either the developers who conflict often just do not communicate well enough, or they have overlapping responsibilities, and this must be addressed at a human level—no version control system, no matter how smart, can fully automate project management.

Now let's see how to undo changes that have actually made it into the master copy.

We talked earlier about the case in which you accidentally clobbered your own code that had 80% of the functionality complete, and would like to now revert to the version that was mostly working. That was easy, you could discard your working copy, re-update to the master copy, and voilà, you had your 80%-working code back.

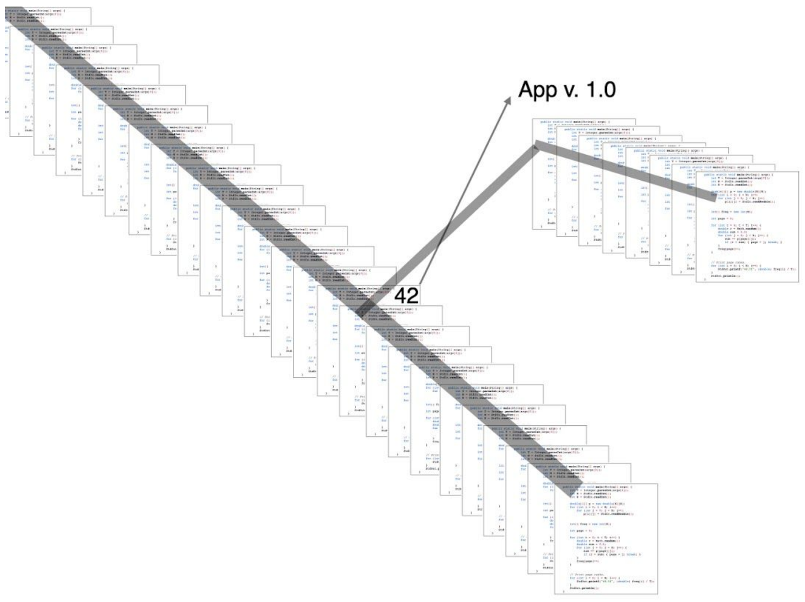
A similar situation can occur if, when Peter attempts to resolve a conflict, he does it wrongly, for instance he inadvertently removes one part of Jane's changes. This time, however, when Peter commits, his changes propagate to the repository into version 41 and, from there, eventually, to every developer's working copy. This ends up breaking the project, and everybody hates Peter for this. To save his honor, Peter retrieves Jane's version 40 from the repository and merges it carefully with the HEAD, and then commits the newly merged state of the code to the repository. Now all developers can pull the new version of the code and be happy. Remember, versions are never updated in place, they are always appended.

Branches

Consider a more sophisticated version of the scenario we described above: Jane and Peter completed the first release of their project and decide to publish it, say in the app store. As it always happens, there were several features that could not be completed on time and are therefore not part of this version 1 of the software. Now users get to play with version 1, while Jane and Peter start working on version 2.

The problem is that users start reporting bugs in version 1, but Jane and Peter have already moved their repo ahead of version 1 to the new version they're working on. So they cannot just fix the bug in v.1. They could perhaps fix the bug in the current v.2 code and deploy that one to the users, but it's quite likely that v.2 has even more bugs than v.1.

The way to handle such situations (which are guaranteed to occur) is to create a "branch" of the project every time there is a release of the software. So, in the sequence of revisions, Jane and Peter would branch version 1.0, ship that to their users, and continue their work on the HEAD. You can think of branches as parallel universes When the branch occurs, there is virtually a complete clone produced, which then takes on a life of its own.

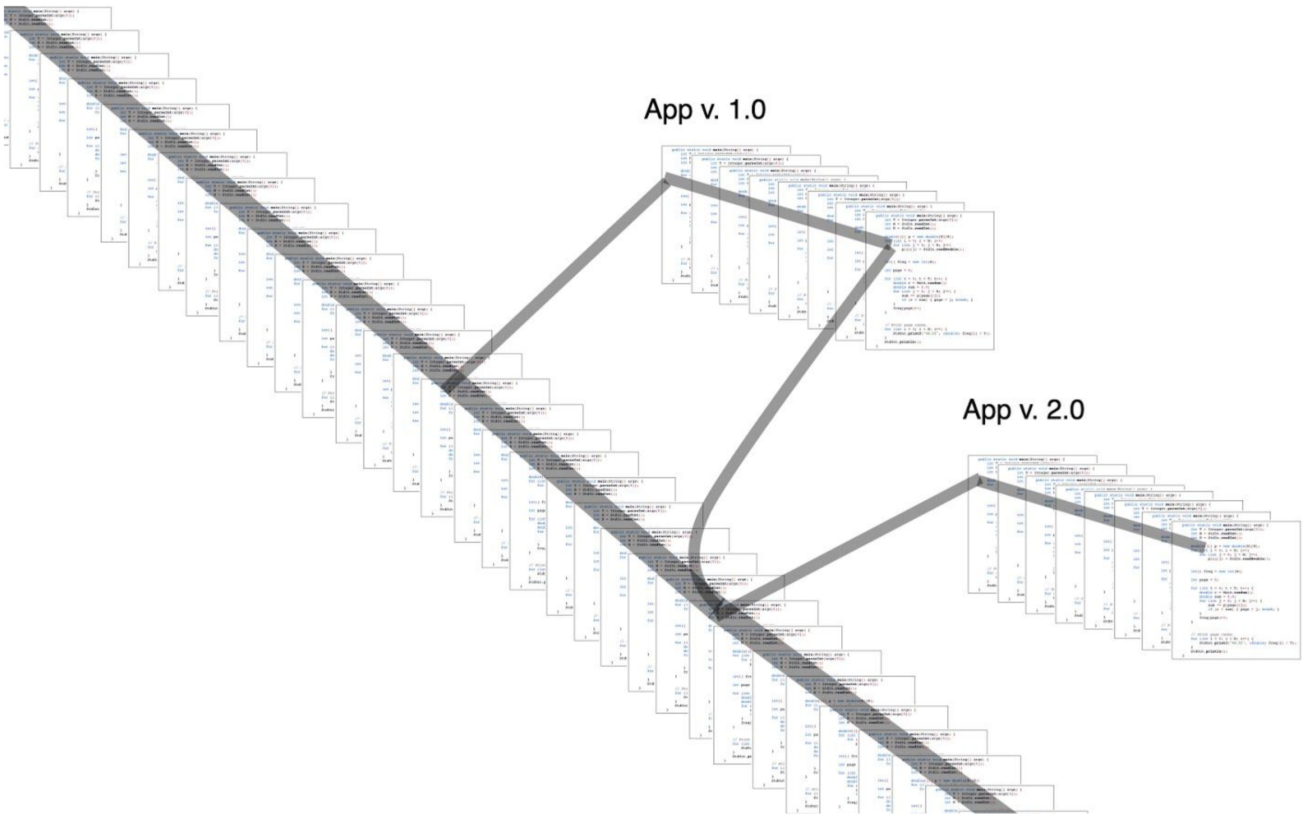


They can now operate with two timelines: bug fixes go into the v.1.0 branch, while new functionality goes into the main line. They can commit changes to either one branch or another. On the v.1.0 branch they would only commit bug fixes, and each such commit would generate a new version of that code, as shown in the above figure. Eventually, this branch will produce a bugfix release, perhaps called v.1.1, which contains no new functionality but only bug fixes on top of version 1.0. This v.1.1 release will then be shipped to users. In parallel, as functionality is added to the HEAD, new versions are created along that main "trunk" of the repository, which is now starting to take on the shape of a tree.

With this process, once Jane and Peter are ready to release v.2, they branch it off from the mainline and proceed with developing v.3.

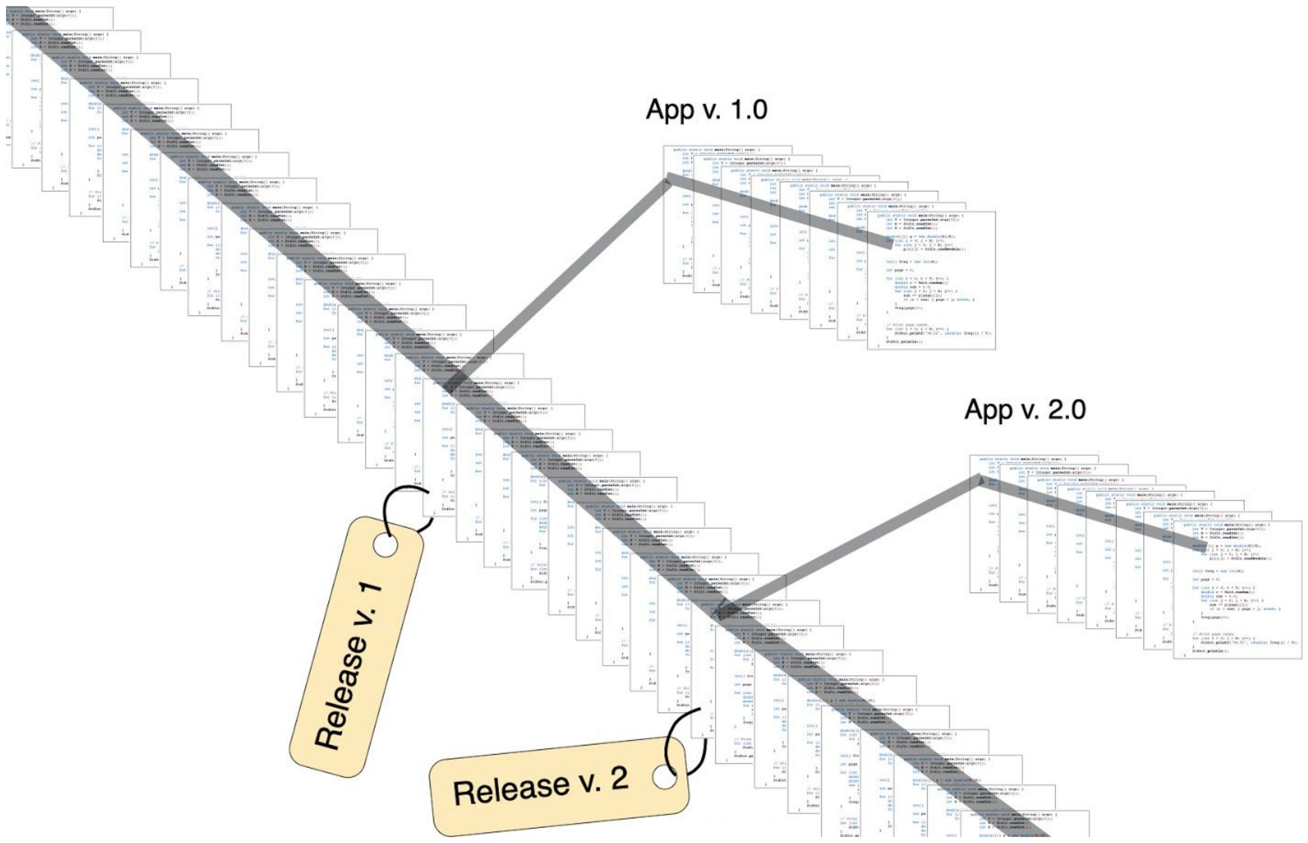
If a bug fixed in the v.1.x branch still exists in the HEAD, then the bug fix may be merged not only into the v.1.x branch but also into the trunk. It is not always the case that the bug exists in the trunk, because redesign or refactoring may have completely eliminated the code that housed the bug, or changes to other parts of the code made that code behavior that was previously buggy be correct now.

Alternatively, release branches that contain only bug fixes could be merged wholesale back into the trunk, in order to bring all those fixes back to the main code. This way, individual merges are avoided. However, such wholesale merges are more complicated, because the number of changes is large and the number of conflicts can be proportionally high. One type of operation called "cherry picking", described further below, helps pick out only the desired commits.



Branches of course do not make the problem of merging go away: you still have to reconcile different versions with each other.

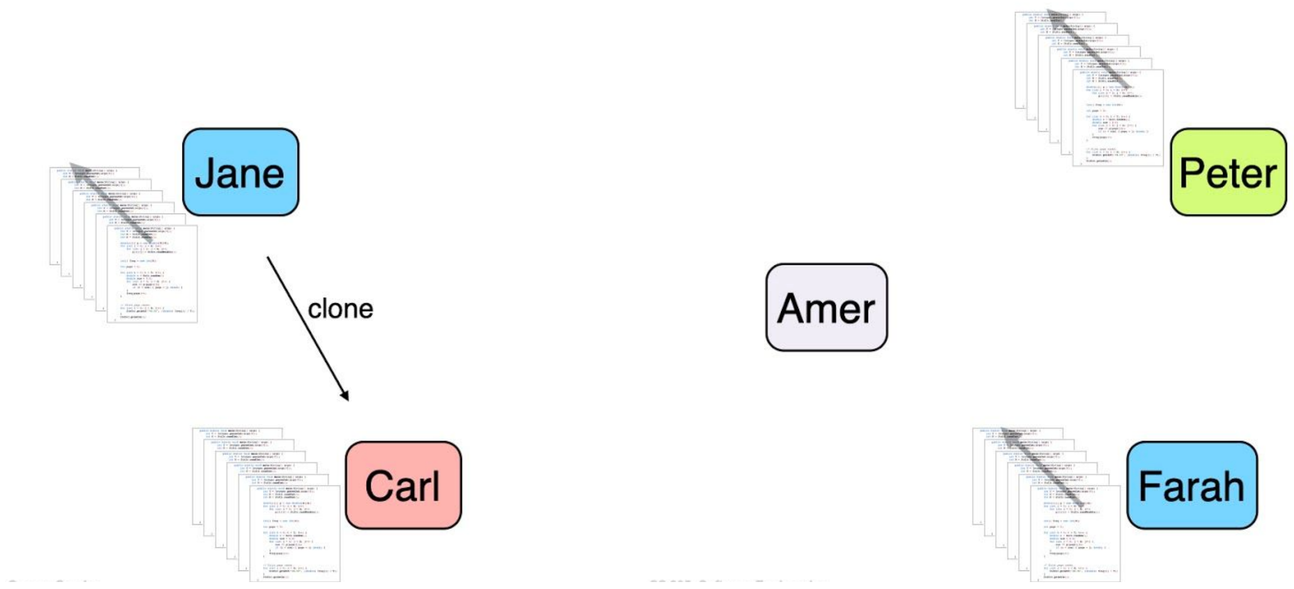
As you might imagine, the number of commits in a project can be large. It's quite common to have thousands of commits, and in projects like the Chrome browser there are more than a million. So navigating the history becomes quite difficult. This is why most VCSs allow you to "tag" specific points in history, in essence giving them a particular name. For example, Jane and Peter may want to tag the point at which they branched version 1 of their app as "Release v.1". When they branch v.2, they would tag that point in the timeline as "Release v.2". This makes it easier to search for and find these points in the timeline.



This is the gist of second-generation version control. One thing that is fundamental to this model is the idea of there existing a centralized server that hosts the repository and has the master copy.

Modern version control systems

Modern, third-generation version control systems, such as Git, take a peer-to-peer approach to version control. Instead of there being a single, central repository with which Jane and Peter synchronize, each developer's working copy of the project turns into a repository. To synchronize repositories with each other, developers exchange sets of changes (also called patches) among themselves. In some sense, the master copy now becomes virtual, and it is replicated or distributed across several working copies.



Although not much changes fundamentally, there are some important differences that such a distributed VCS (also called DVCS) introduces:

First, there is no master copy anymore, which means that each working copy carries with it the entire history of the project. A developer can now examine the entire timeline without having to be connected to the master repository, or not having to be connected at all to a network.

Second, what was "working copy" before is now a "staging area": Commits don’t go to a central repository but to the local repository. When you add files to your staging area, you are essentially marking them as members of your next commit, which will then add them to the snapshot of your project as reflected in your local repository. So the workflow in such a VCS consists of modifying files, then staging them (meaning that you add snapshots of them to your staging area), and then committing them, which takes those snapshots and stores them permanently in your local repository.

Third, if Carl wants to get a copy of Jane's repo, to contribute to it, he gets what is called a "clone" of Jane's repo. By doing so, Carl receives a copy of virtually all data that exists in Jane's repo, including every version of every file in the project's history. Developers' individual working repos now become full-blown backups of the repository, and every new clone in effect causes a new backup. In Git, any of the repos could serve as remotes, thus becoming a common repo that the entire team pushes to/pulls from their changes.

Of course, it is possible to have a remote repository that is made available to other developers, so that you don't need to export your own repo to anyone who is interested; a service like GitHub provides this kind of sharing of remote repositories. So, for instance, Carl could have gotten a clone of Jane's repo from GitHub, and then push and pull data to it as needed.

Fourth, branching is a lot more frequent in a DVCS like Git. Remember, in second-generation VCSs, branches marked a divergence from the trunk (i.e., the main line of development) enabling work to be done (such as fixing bugs) that does not interfere with that main line (which was v.2 in our example). A modern distributed VCS encourages a workflow in which branching and merging is done considerably more frequently than in a classic VCS, in recognition of the fact that developers often multitask between issues they are working on. Correspondingly, the notion of trunk starts to lose its meaning, as it merely becomes a branch itself, often referred to as the "master" branch (or, in the near future, "main" branch).

Remember that we advise to not combine a bug fix with some new functionality in the same commit. To keep these separate, you can branch from the master branch, add the functionality on one new branch, add the bug fix on another new branch, and eventually merge them both back into master. While developing, you can move back and forth between the branches, commit changes separately, and generally manage the two new"parallel universes" separately from each other. You no longer have to work on the two sets of changes serially, you do them in parallel, on separate branches.

On Github, a popular repository hosting service, there exists the notion of a "pull request", or PR for short. In the model where multiple developers share a hosted repository, a PR is a way of telling the other developers that you have a new set of changes in a branch of the repo. Other developers can then review your proposed changes, they can add review comments, and even add commits to that PR. You yourself can also add follow-on commits. In this way, everyone who is interested can contribute their knowledge to these changes. Once happy with the set of changes contained in the PR's commits, a maintainer of the shared repo can decide to merge the PR into the corresponding target branch.

It is worth noting that not all files in a project are worth tracking versions of. For instance, there may be temporary files Jane has created, or backup files, or even intermediate files from the build process, and these are not at all useful to Peter, and it's unlikely you'd ever want to go look at a previous version of these. This is why every VCS allows you a way to specify which files should be tracked -- and therefore stored in the repository -- and which should not. A file that is committed to the repository must have been added to the working copy at some point.

This is, in a nutshell, how version control systems work. VCSs, whether centralized or distributed, are absolutely essential to developing software with more than a handful of developers. Version control is also known as revision control or source control, so if you hear these terms, don't think they're any different. Version control systems are part of a broader set of tools, called software configuration management, or SCM tools for short, which have become a key tool for enabling the construction of increasingly more complex software systems.

Example workflows

Feature branches

If you want to work on a self-contained unit of work, such as a new feature or a fix for a specific bug, version control allows you to work independently of the other code, and to encapsulate the resulting code in a way that is easy for others to review.

Create a branch, e.g. `git checkout -b feature/cloud_backups` for a branch in which you'll add a "backup user data to cloud" feature. Note that the slash '`/`' is just another character in branch names, it has no special meaning, but it allows you to clearly state what a branch is for. You could for instance name a branch `bugfix/disappearing_button` if you are fixing a bug in which a button disappears. Branch names are part of the codebase, so name them with the same care you use when naming variables or classes.

Then, work on the branch, creating a commit for every time you complete a meaningful unit of work.

Once you're done, you can present this branch to your teammates, for instance using a pull request. Once they commented on and reviewed your code, and everyone is satisfied, this branch can be merged into the codebase.

The main advantage of the "feature branch" workflow is that it provides atomicity of changes: no matter how long it takes to write the code, even if you take a break to work on something else, either all of the feature is in the codebase or none of it is, thus your teammates will never see "half-working" code.

"Cherry-picking" commits

Imagine you are developing some code in a branch when you stumble upon a particularly problematic bug (e.g., it can lead to important data loss). Your team agrees it needs to be fixed as soon as possible, but your branch is not finished. That's where "cherry-picking" comes in: you can apply a commit made in one branch to another branch, without merging the other commits in that other branch. For instance, if you just made the fix in branch `A`, creating a Git commit with hash `abc123`, and you'd like to apply this commit to branch `B`, switch to branch `B` using `git checkout B` then `git cherry-pick abc123`. This will add a new commit on branch `B`, with a different hash from but the same contents as commit `abc123`, with no other changes.

You can also use cherry-picking to resurrect old commits from branches that were never merged. For instance, perhaps a few months ago some branch `X` was abandoned because priorities shifted in your project, but as part of the code on that branch there is an implementation of some utility code that is now needed for some other purpose. Use `git log X` to show the commits from branch `X`, which you can then cherry-pick. Just like merging, Git may ask you to manually resolve conflicts.

Rewriting history in Git

Git allows you to rewrite history, i.e., the log of commits that is normally append-only can be manipulated to make it look like something happened in the past. For instance, you could fix an incorrect commit message from an old commit, or "squash" two commits together if they make more sense as one.

If you're interested, look up the `git rebase` and `git push --force` commands.

However, rewriting history is dangerous because unlike other Git operations, it can lead to data loss. For instance, you could accidentally tell Git to forget about an old commit while you're modifying others, and then that commit is gone! We recommend that if you rewrite history, you should always first push to a remote repository such as GitHub, then do the rewriting, double-check that the result is exactly as you want it to be, then force-push.

Five rules for commit messages

The sequence of commit messages in a version control system forms a sort of diary of the development project. Think of this diary as write-once/read-many, i.e., whatever goes in is written once, and is then read many times. The commit log is a powerful tool for understanding how a body of code has evolved..

Since commit messages are so important to navigating the timeline of a project, it is crucial that they be descriptive. In fact, commit messages are as important as good comments in code. A good commit message conveys clearly who did what and why. You should think of commit messages as complementing the code's history and code comments. A message that says "Fixed some bugs" conveys hardly any information, but if instead it says "Protect with a lock the connection list", it suddenly becomes much more informative. The worst are empty commit messages or wrong commit messages that deceive other developers.

In the specific case of Git, the VCS we use in this course, if you follow the following 5 rules, then both you and your fellow developers will derive a lot more use from the version control system.

1. Separate subject from body with a blank line.
2. Limit the subject line to 50 characters.

Did you know that a commit message in Git has both a subject and a body? Not every commit requires both a subject and a body (in fact, most of the time, you'll find that a 1-line subject is enough). Nevertheless, when you use both, separate them by a blank line. This separation of subject from body pays off especially when browsing the log, e.g., with `git log –oneline`.

The subject line itself is good to keep no longer than 50 characters. This is not necessarily a hard limit, it's just a rule of thumb for a length that keeps the subjects readable. If you use GitHub, you will see that it truncates the commit subjects to 72 chars. If you find it difficult to summarize what the commit is about in 50 characters, then think very well whether you are not committing too many changes at once.

To illustrate these two rules, compare this:

```
$ git log --oneline -5 --author cbeams --before "Fri Mar 26 2020"
e5f4b49 Re-adding ConfigurationPostProcessorTests after its brief removal in r814. @Ignore-ing the
testCglibClassesAreLoadedJustInTimeForEnhancement()
2db0f12 fixed two build-breaking issues: + reverted ClassMetadataReadingVisitor to revision 794 +
eliminated ConfigurationPostProcessorTests
147709f Tweaks to package-info.java files
22b25e0 Consolidated Util and MutableAnnotationUtils classes into existing AsmUtils
7f96f57 polishing
```

to this:

```
$ git log --oneline -5 --author pwebb --before "Sat Aug 30 2020"
5ba3db6 Fix failing CompositePropertySourceTests
84564a0 Rework @PropertySource early parsing logic
e142fd1 Add tests for ImportSelector meta-data
887815f Update docbook dependency and generate epub
ac8326d Polish mockito usage
```

Discuss with your team the style to be used for commit subjects and bodies (e.g., length of lines, capitalization, punctuation) as well as metadata (e.g., how to reference tracking IDs for issues, pull requests, etc.) If everyone on the team does it consistently, it’s easier to parse commit histories. Remember, the commit log is a journal

3. Use the imperative mood in the subject line.

The imperative mood makes communication very clear, since (by definition) it expresses who needs to do what. Think of the Git commit message subject as filling in the blank in the sentence "If applied, this commit will ..." An added benefit of this formulation is that, at least in English, it saves quite a few characters.

Here are a few good examples:

`Refactor subsystem X for readability` → *If applied, this commit will refactor subsystem X for readability*

`Update getting started documentation` → *If applied, this commit will update getting started documentation*

`Remove deprecated methods` → *If applied, this commit will remove deprecated methods*

4. Wrap the message body at 72 characters.

In the old days, alphanumeric displays used to be 80 characters wide. Even though that is no longer the case, 80 seems to be a sort of natural width of a line of text that is comfortable to read before you need to move your eyes.

Plus, Git likes to do some indentation of the text for the various log inspection commands, so if you keep the lines no longer than 72 characters, then you will always have a nicely formatted output.

5. Use the message body to explain *what* and *why* not *how*.

The body of a commit message can provide valuable context that can save fellow and future committers lots of time in understanding what the commit did. There is no need to describe *how* something is done, since that is contained in the code. The real question is *what* exactly has been done, at a high level, and *why*. In many cases, in the absence of context, the commit can be misinterpreted and leads to the introduction of bugs by developers who got the *what* and/or the *why* wrong. So use the message body to explain the reasons why you made the change in the first place—the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did.

These five rules are not hard and fast. It's just that the more consistent you are in their use, the easier it will be for you to work with others.

Here are the rules in a nice box for you to take along:

The 5 rules of commit messages

1. Separate subject from body with a blank line.
2. Limit the subject line to 50 characters.
3. Use the imperative mood in the subject line.
4. Wrap the message body at 72 characters.
5. Use the message body to explain *what* and *why* not *how*.

Building code

When the time comes to ship a software product to a client, a development team needs to compile every source file, resource and dependency into a single package containing one or multiple binary artifacts. This is a critical and non-trivial step in the development process. Indeed, the number of files involved can be very large (the Linux kernel alone was composed of around 62’300 source files at one point) and for every compiled module, dependencies need to be resolved, compiled and packaged. Modules, as well as their interactions, need to be tested and new versions need to be tested against older ones to check compatibility. On top of that, the code may need to run on multiple platforms (Windows, MacOS, Android, iOS, others…) which need to be treated individually.

To help in this task, developers use a **build system**, which automates all stages of the process of transforming a collection of source files into a shippable artifact. As this operation should be run on a commit basis, which is very frequent, these tools need to be very efficient and reliable.

File organization

The first step is to make the code easily manageable by organizing the codebase's file structure. A good file tree structure should make it easy to understand the overall project architecture so that, if a bug occurs in a certain module, the files containing the problematic code can be easily tracked down. It also helps the build system by grouping files by compilation unit.

Unfortunately, no single solution exists for this problem. It is very much dependent on the language/framework used. Some compilers expect a certain file structure, some let the user deal with it entirely. Fortunately, conventions and good practices exist for almost every language and framework, and using them is a good first step into having a good file structure.

Example: Java – This is an example where the file structure is enforced by the compiler. In Java the file structure reflects the package structure defined in the code. This makes it very easy to find a specific file but this can also lead to very deep directory structures (ch.epfl.sweng.project.module…)

Example: C/C++ – In C/C++, the file structure is entirely created by the developer. In this case, public headers and the source code are usually stored in separate directories. This is done specifically because of how libraries are exported. The source files are compiled into a `*.lib/*.dll` file that contain the executable and the public headers define the interface available in a specific library.

Example: Android – In Android projects, source files that deal with business logic and bindings to UI are located in app/src/main/java/<custom package path>/ (however, more packages should be appended to separate Activities, data models, service providers, ... more on that in the lecture about design patterns and MVP). Instrumented tests that run on an device emulator live in app/src/androidTest/java/<custom package path>/, unit tests in app/src/test/java/<custom package path>/, UI views in app/src/main/res/layout/ and other resources reside in the app/src/main/res folder.

Compiling

The next logical step is to compile your source code. However, most programmers don't want to invoke the compiler on every single source file so we can again make use of the build system to automate this task. Some languages/frameworks already have great tools, you have surely noticed compiling your Android Studio project is as simple as clicking a button. Some lower level languages like C/C++ delegate this complexity to the user. Fortunately, there exists a lot of scripting tools, like CMake and Make to automate the build.

The goal of a good build system is to efficiently compile a large amount of source files. It should also report compilation errors to the user along with all the necessary information to fix the error quickly. Creating build scripts should be straightforward: if the complexity of a build system is greater than the complexity of the code, it would be a waste of time. It should also let the user compile code for multiple platforms. Languages like Java don't have this issue as it is compiled to run on the JVM, but most other languages don't. Fortunately, most build systems like Premake have built-in support for platform switching so developers don't have to write multiple versions of the same build script for each platform.

Of course, the build system needs to be run on instances of each different platform to be able to compile to them (you cannot compile for macOS on a Windows machine). Developer teams often rely on build servers that run instances of all the target platforms through virtualization or containerization. This enables projects to be compiled for all platforms in parallel, saving a considerable amount of time.

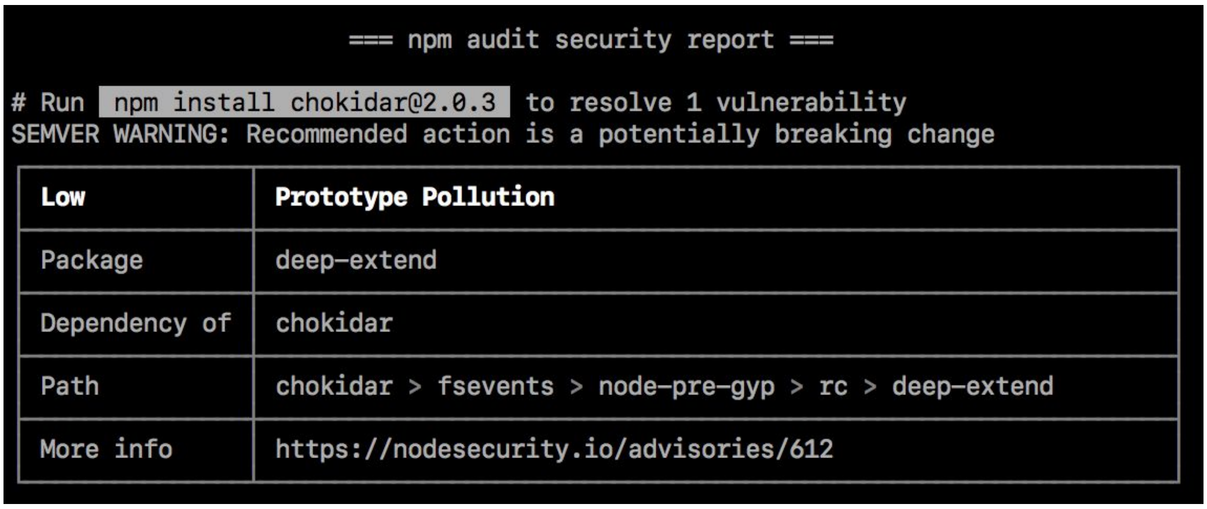
The build process can also include unit tests. These tests validate the basic functionality of the program and are the first line of defense against bugs. However, these are run "offline", meaning they do not interact with the outside world (Web API, Database, …) but rather a fake, predefined version of it called a mock. This means that they do not entirely prevent the apparition of bugs in production. You will learn more about testing later in the course.

Dependencies

As was mentioned in previous chapters, modularity helps with code complexity and code reuse. Indeed it would be a waste of time to write and test a functionality that has already been written and field tested by someone else. Of course, using external code can introduce instabilities and tracking down bugs that come from dependencies and finding a workaround can cost a lot of time, so choosing a 3rd party library should be done carefully. Thankfully, the Internet provides a lot of places where developers can share their work which has greatly increased the variety of available tools (cf. *The Open-source Movement*).

Large projects, however, can depend on a lot of third party code that needs to be integrated in the build pipeline and this, depending on the language, can be a challenging task. For example, dependencies for C/C++ need to be either built from source or the library for the correct platform needs to be manually downloaded and linked in your final assembly. Other languages, like Rust and Go, have specific ways of dealing with dependencies that are enforced by the compiler itself (this is a common trend for more modern languages) that mostly automate this task. Today, most programming languages have tools that provide easy access to dependencies (vcpkg for C/C++, Maven for Java, …).

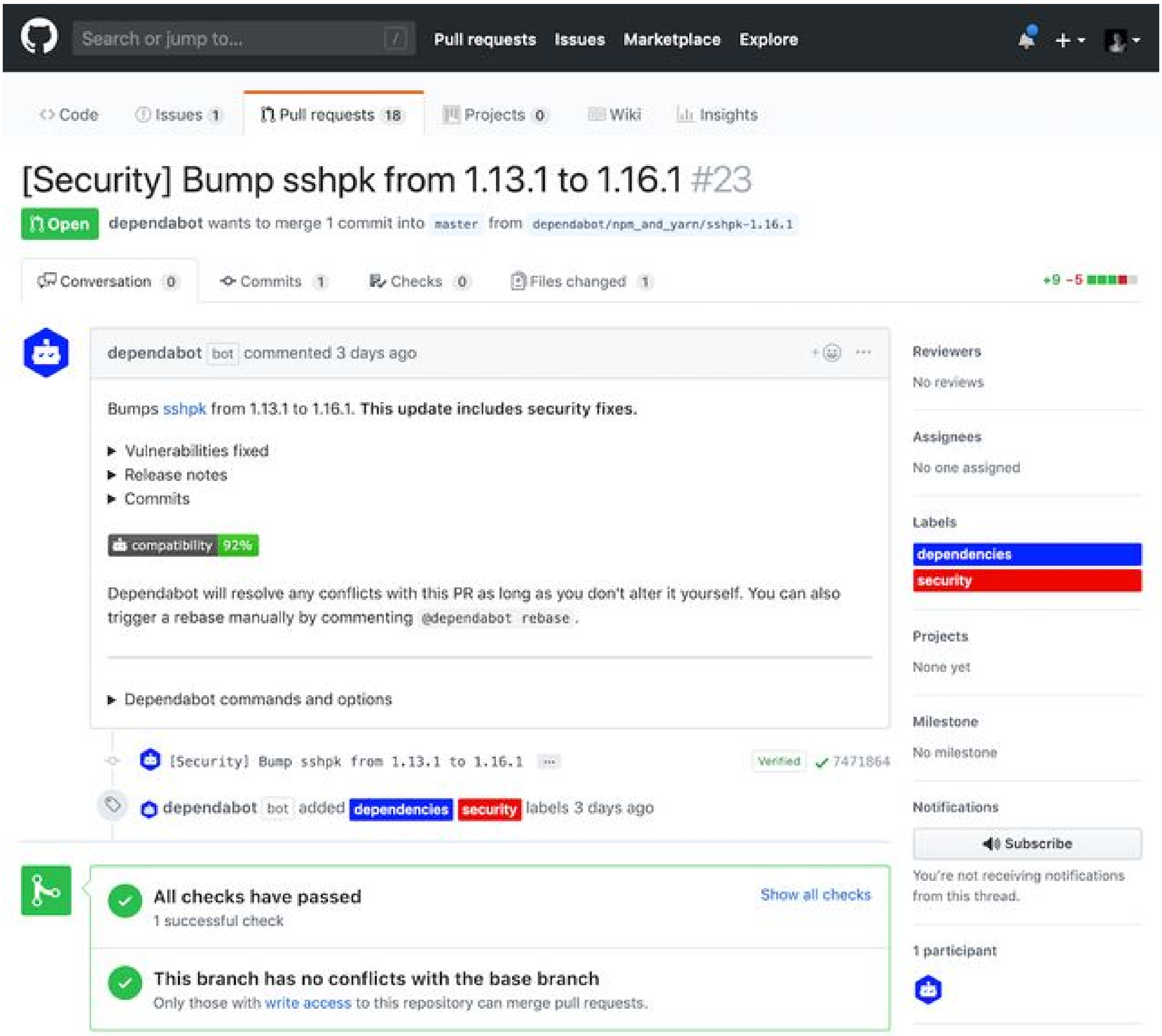
Another common problem with dependencies is dealing with multiple versions. When a new dependency version is introduced, it needs to be tested and validated to ensure that the build will remain stable once it is updated. This process can be very time consuming, especially for older languages or if new versions of libraries are often available. However, modern build tools also help with this task, as they usually bundle some commands to automatically scan and update the dependency tree of a project (as well as track libraries with security vulnerabilities). The continuous integration pipeline can also integrate tools dedicated to watching package registries, automatically updating the dependency tree and generating a pull request on a new branch when a new version is released; dependabot on Github is an example of such a tool.



```
                    === npm audit security report ===

# Run  npm install chokidar@2.0.3  to resolve 1 vulnerability
SEMVER WARNING: Recommended action is a potentially breaking change
┌───────────────┬──────────────────────────────────────────────────┐
│ Low           │ Prototype Pollution                              │
├───────────────┼──────────────────────────────────────────────────┤
│ Package       │ deep-extend                                      │
├───────────────┼──────────────────────────────────────────────────┤
│ Dependency of │ chokidar                                         │
├───────────────┼──────────────────────────────────────────────────┤
│ Path          │ chokidar > fsevents > node-pre-gyp > rc > deep-extend │
├───────────────┼──────────────────────────────────────────────────┤
│ More info     │ https://nodesecurity.io/advisories/612           │
└───────────────┴──────────────────────────────────────────────────┘
```

Example run of npm audit, which scans NodeJS projects for vulnerable libraries



[Security] Bump sshpk from 1.13.1 to 1.16.1 #23

Open dependabot wants to merge 1 commit into `master` from `dependabot/npm_and_yarn/sshpk-1.16.1`

Conversation 0 · Commits 1 · Checks 0 · Files changed 1 · +9 −5

dependabot bot commented 3 days ago

Bumps sshpk from 1.13.1 to 1.16.1. **This update includes security fixes.**

▸ Vulnerabilities fixed
▸ Release notes
▸ Commits

compatibility 92%

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

▸ Dependabot commands and options

`[Security] Bump sshpk from 1.13.1 to 1.16.1` Verified ✓ 7471864

dependabot bot added dependencies security labels 3 days ago

All checks have passed
1 successful check
Show all checks

This branch has no conflicts with the base branch
Only those with write access to this repository can merge pull requests.

Reviewers: No reviews
Assignees: No one assigned
Labels: dependencies security
Projects: None yet
Milestone: No milestone
Notifications: Subscribe
You're not receiving notifications from this thread.
1 participant

An automatically generated pull request by dependabot on GitHub

Packaging and versioning

This last step combines all binary artifacts, resources and dependencies that a product needs to fully run in a single package (installer, archive, …). To minimize the size of the output, it only needs to include what is strictly necessary. The package is assigned a unique version number (conventionally follows a pattern of the form `MAJOR.MINOR.PATCH`, more information on semantic versioning if you are interested). It can then be sent to the client to be deployed in production.

Tagging a package with a version number can also be very useful when a client reports a bug on an older version of your code. It allows the developers to roll back to the specific environment that triggered the bug and try to reproduce it. This can often lead to the discovery of sneaky bugs that only occur in certain, very specific, scenarios (dependency or driver incompatibility, …).

Conclusion

The build system is the central piece that allows you to convert all your source files into a single package that can be shipped to the client to be deployed. A good build system should provide a short time-to-delivery and greatly reduce the occurrence of bugs in production. But a single run of the entire build pipeline can still take a long time and having the main branch fail to build can mean a few hours to a full day of delay. To avoid this situation, teams need a workflow that makes this situation impossible to occur in the first place (e.g., feature branches and merge requests). These workflows need to be enforced as they are only useful if followed. This is where it all comes down to the developer's individual discipline.

The directed acyclic graph of tasks for the Java Gradle plugin (each arrow designates a "depends-on" relationship). "compileJava" and "compileTestJava" build the code into Java bytecode, while "test" triggers the unit tests. "check" can also run optional tools, such as the JaCoCo test coverage analysis.

The “compileJava” task can be further broken down into compilation steps for building dependencies (green tasks)

Continuous Integration

Today, software products are no longer developed by one person alone but by teams. These have various sizes from two to thousands of developers working together on the same project. In this kind of setup, many things can go wrong and it becomes crucial to identify problems as early as possible, otherwise they become difficult and expensive to fix.

For example, when starting to fix a bug or add a feature, a developer takes a copy of the shared code base. As other developers submit changed code to the repository, the developer's copy gradually ceases to reflect the repository code; new code gets added as well as new libraries, and other resources that create dependencies, and potential conflicts. The longer development continues on a branch without merging back to the mainline, the greater the risk of multiple integration conflicts and failures when the developer branch is eventually merged back. When the developer finally attempts to merge the code she wrote, she can encounter problems – the more changes the shared repository contains, the more work is involved in submitting the changes.

The basics

Continuous integration (CI) is the practice of merging all developers' working copies to a shared mainline (could be a dedicated branch in the repo) several times a day and automatically building the software, allowing teams to detect problems early. That is why this integration is called continuous. By integrating so frequently, there is significantly less back-tracking to discover where things went wrong, so you can spend more time building features and less time debugging. Continuous Integration doesn’t get rid of bugs, but it does make them significantly easier to find and remove. The quicker a bug is discovered, the easier it is to fix.

The first step in CI is to maintain a *single source code repository*, managed by a version control system, and all artifacts required to build the project (code, configuration files, etc.) go into this repo. The entire system should be buildable from a fresh checkout and not require any additional dependencies outside the repo. The mainline (or master in Git parlance) should be the place for the working version of the software.

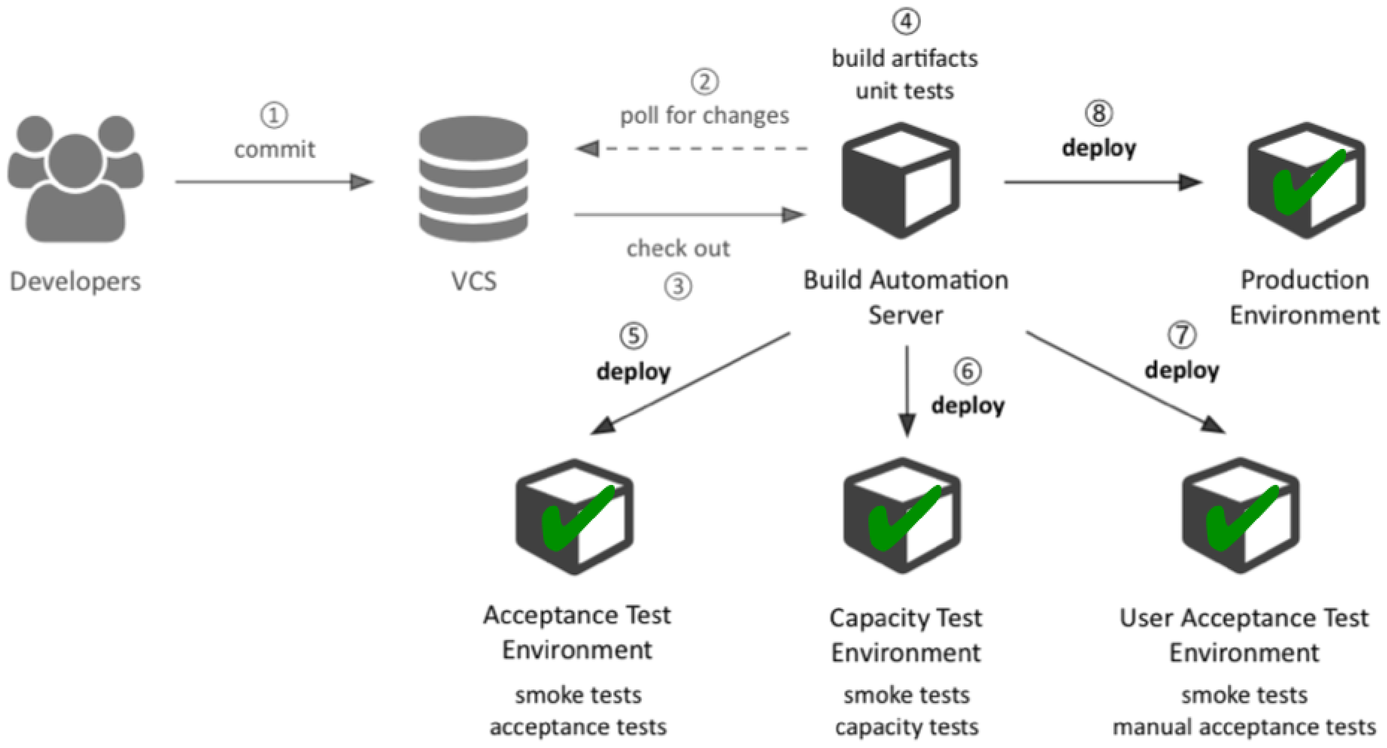
The second step is to *automate the build and test process*, i.e., a single command should build the whole system (e.g., make, Gradle). Automating the integration part often includes deployment into a production-like environment. The build script may not only compile binaries but also generate documentation, website pages, statistics and distributions for specific platforms (such as Debian DEB, Red Hat RPM or Windows MSI files). Once the code is built, the test suite is automatically invoked and runs autonomously start to finish.

Third, *every commit to mainline has to be built*, i.e., run an integration build on every commit. The CI server usually does this automatically. Since you're building every commit, it becomes easy to track down problems through a granular diagnosis: after which commit did the build start failing? That would be the first place to look for the problem. Ideally a failing commit should not end on master before the problem being solved. This would indeed be against the principle introduced before: master should be the place of working software. That being said, some projects can become so big that building them takes several hours. Microsoft Windows, for example, takes around 16 hours to build so building it on every commit is out of reach at this point. To mitigate that, a possible solution is to build automatically at a certain point in time instead. Microsoft adopted this solution for the Windows development process, where the whole OS is built once a day during the night.

Fourth, *keep the build fast*. If the build process completes rapidly, then integration is swift and discovery of problems becomes quick. This requires somewhat of a departure from the standard way of thinking about performance, where we care just about how fast the end product runs -- here we also consider how fast the process of building the product is. For example, a key ingredient in making the builds fast is for the test suite that runs automatically to be lightweight and run quickly. This can be quite tricky for projects that require UI tests suite instead of only unit tests. To keep the build fast, one must also think about the architecture of the build infrastructure: considering parallelizing the tests execution, split the jobs across several servers, ...

To increase the chances of finding important problems early, *test in a clone of the production environment*. Test environments often differ dramatically from production environments (e.g., the environment is controlled: the versions of used software are known, there are no or few other applications installed that can interfere with yours, the configuration of the machine is also known, data is controlled and not the user's), and many failures manifest in production but not in test environments. For example, in 2018, a bug in the Uber application caused notifications to arrive on a phone on which someone was once logged in but has been logged out since; this can pose serious privacy problems. The test environment used for testing in your CI should be as similar as possible to the production environment. Such a replica might entail having multiple technology stacks, multiple screen sizes, various brand's OSs (e.g., in the case of Android), various OS versions, various browsers (in the case of a website), etc. Today virtualization technologies lower the cost of building such environments: You can either create some virtual machines and install everything needed on them or clone virtualize some "real production machines". Once you have your set of VMs, it is easy to run them and run tests of the same hardware. You do not need to have several real machines to run in different environments. Moreover, you can destroy the VM used after each run and always start with the same fresh image you prepared.

Related to testing in a production clone is to *automate deployment*, i.e., write a script to deploy your app to a live test server that everyone can look at and use. You can think of this as continuous deployment. The idea behind automated deployments is that manual ones are really slow and unreproducible. Therefore it would be safer and quicker to automate them. The automated deployment extends behind the testing as it is meant to deploy to production environments too after passing through several levels of acceptance.



This diagram shows an example of a continuous deployment pipeline. The first part until the build automation server is the same as in the standard continuous integration scheme. Once a build passes all tests on the build server, it is deployed on the first test environment where some scripts prepare it to be very close (not necessarily identical) to the production environment and install the application. The application is executed to verify that everything is up and running and some automated tests are run to be sure the application conforms to customer important acceptance criteria. The next stage is very similar to this one but the tests performed are oriented to load capacity (e.g., throughput, latency, load stress). The last stage is where the customer performs manual acceptance tests. Then the application is deployed in production (automatically in the case of continuous deployment or after manual intervention when the customer is ready in the case of automated delivery).

The seventh aspect is to make sure that *all developers have access to the latest build* and the corresponding metrics, such as which tests failed. When builds are readily available to the entire team, it becomes easier for the culprit to see the impact of their mistake and understand how to fix it, while also applying psychological pressure to not push code to the shared repo without carefully testing it.

The typical continuous integration workflow is as follows:

- Developers check out code into their private workspaces
- When done, commit the changes to the repository
- The CI server monitors the repository and checks out changes when they occur
- The CI server builds the system and runs unit and integration tests

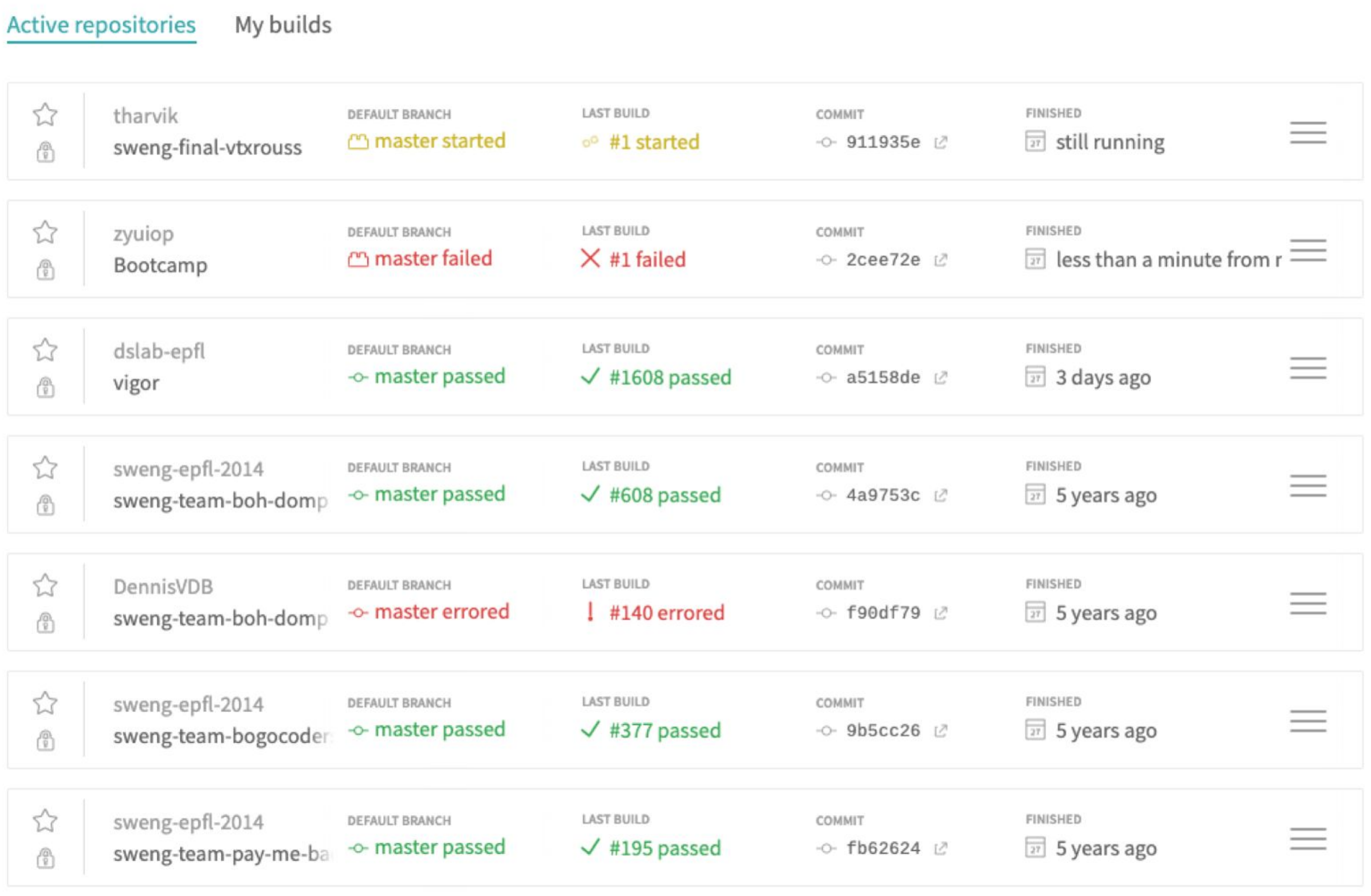
- The CI server releases deployable artefacts for testing
- The CI server assigns a build label to the version of the code it just built
- The CI server informs the team of the successful build
- If the build or tests fail, the CI server alerts the team
- The team fixes the issue at the earliest opportunity
- Continue to continually integrate and test throughout the project

The essence of CI is to establish baselines and make it easy to go back in time to see if and how builds broke. If something goes wrong, this makes it easy to figure out when it went wrong, pinpoint which change broke things, and fix it.

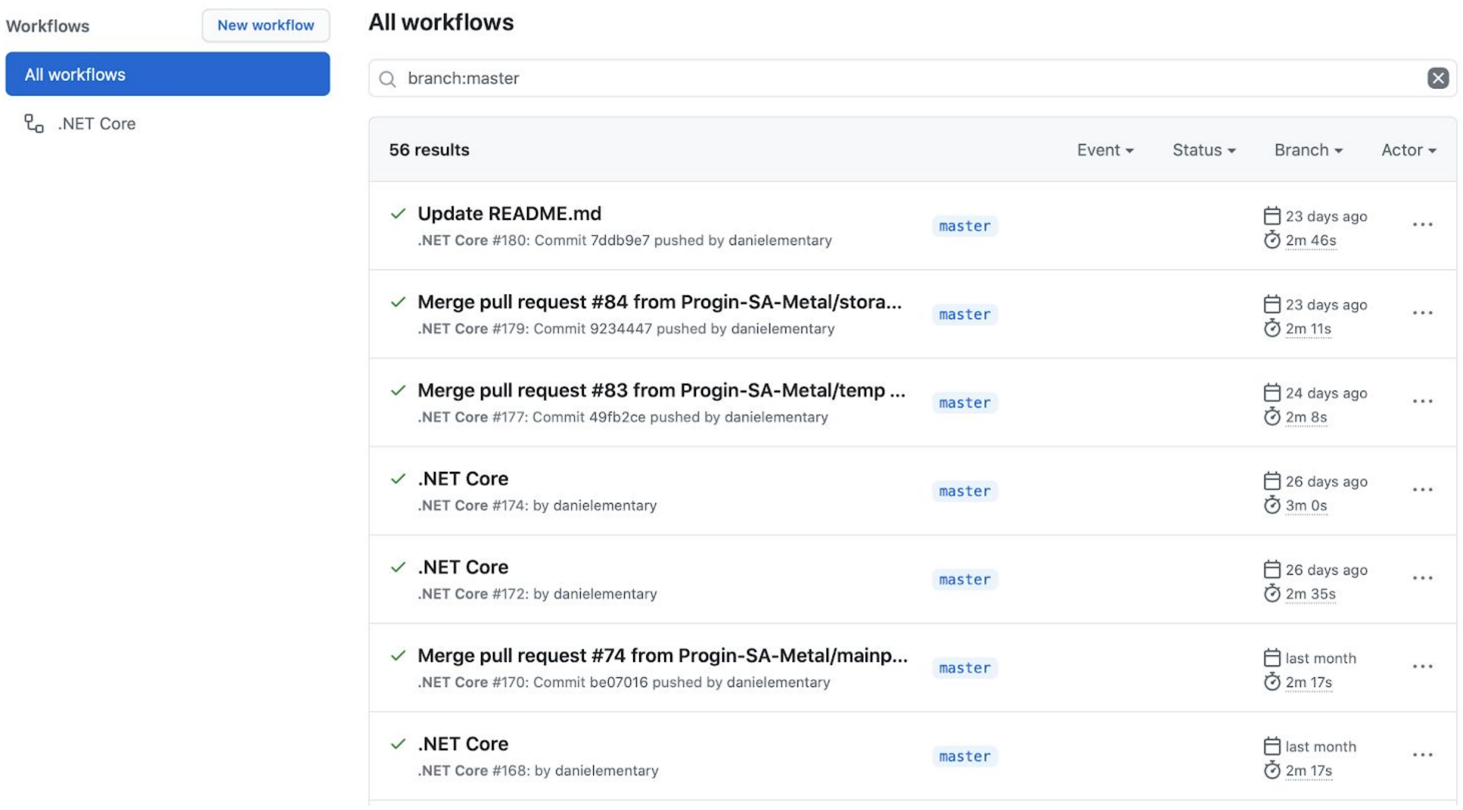
To conclude, here are 5 rules to make CI work best for your team:

- Push frequently
- Don't push broken code
- Don't push untested code
- Don't push when the build is broken
- Don't go home after pushing until you confirm that all is OK

The dashboard of your CI system is very important. Here are some examples.



	DEFAULT BRANCH	LAST BUILD	COMMIT	FINISHED
tharvik sweng-final-vtxrouss	master started	#1 started	911935e	still running
zyuiop Bootcamp	master failed	#1 failed	2cee72e	less than a minute from r
dslab-epfl vigor	master passed	#1608 passed	a5158de	3 days ago
sweng-epfl-2014 sweng-team-boh-domp	master passed	#608 passed	4a9753c	5 years ago
DennisVDB sweng-team-boh-domp	master errored	#140 errored	f90df79	5 years ago
sweng-epfl-2014 sweng-team-bogocoder	master passed	#377 passed	9b5cc26	5 years ago
sweng-epfl-2014 sweng-team-pay-me-ba	master passed	#195 passed	fb62624	5 years ago



Workflows | New workflow

All workflows

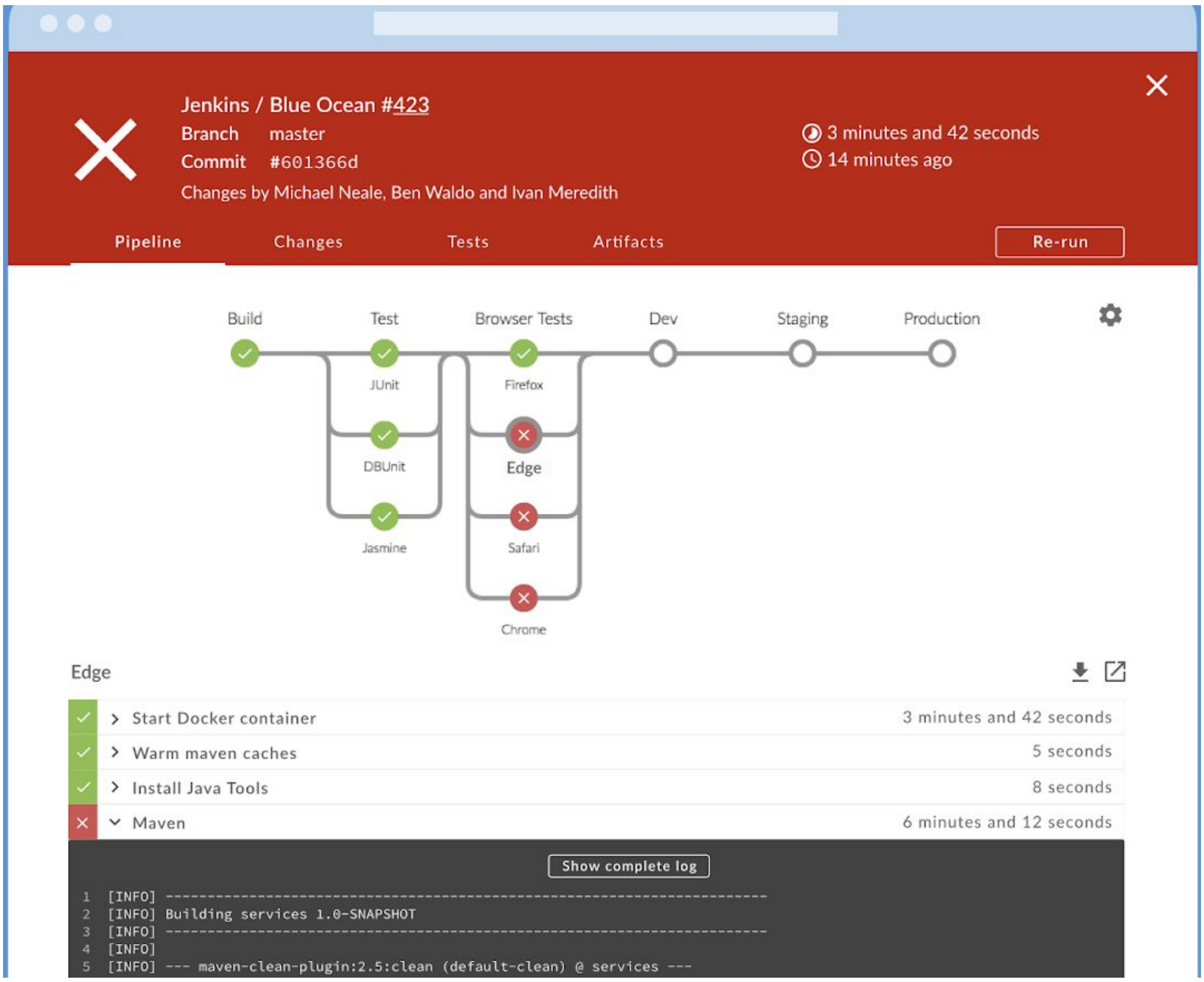
.NET Core

All workflows

branch:master

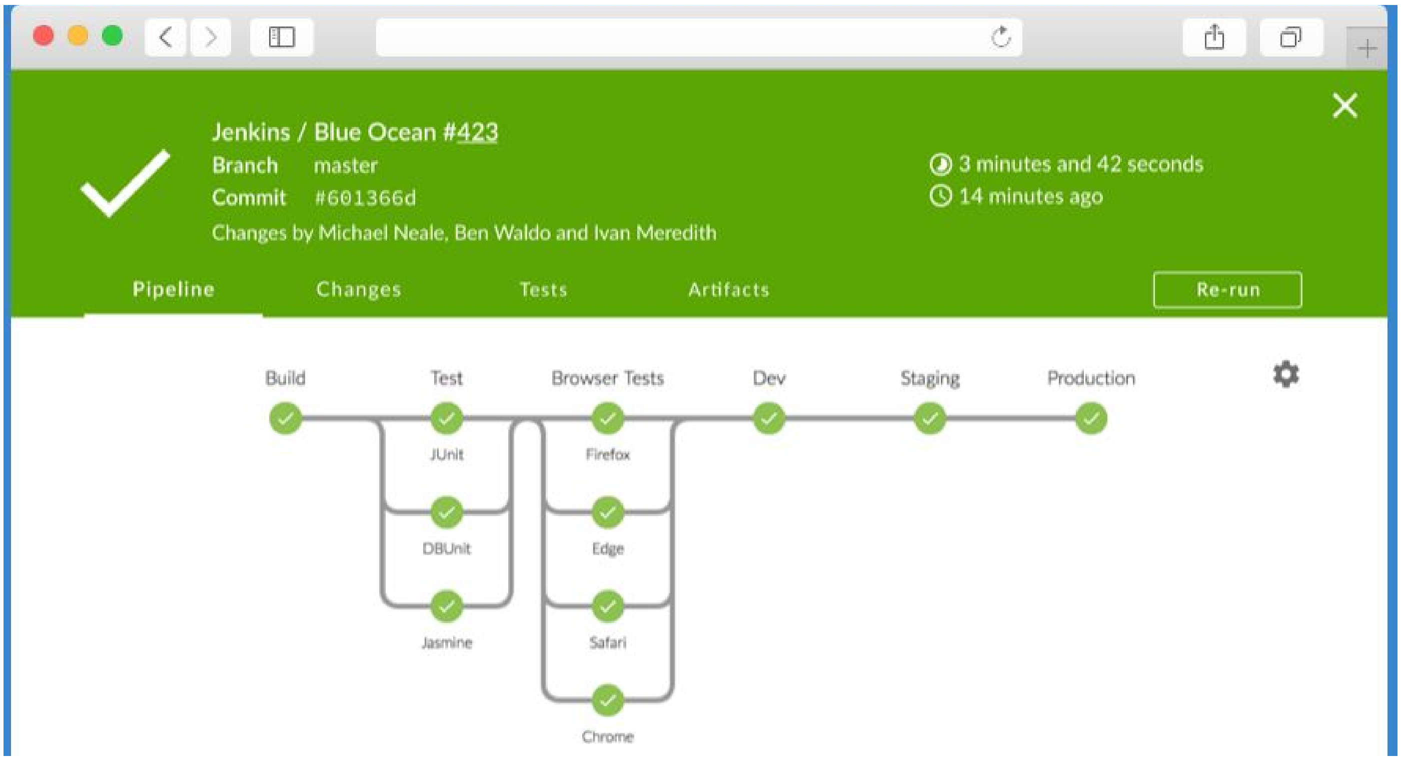
56 results | Event | Status | Branch | Actor

Run	Branch	Date	Duration
✓ **Update README.md** .NET Core #180: Commit 7ddb9e7 pushed by danielementary	master	23 days ago	2m 46s
✓ **Merge pull request #84 from Progin-SA-Metal/stora...** .NET Core #179: Commit 9234447 pushed by danielementary	master	23 days ago	2m 11s
✓ **Merge pull request #83 from Progin-SA-Metal/temp ...** .NET Core #177: Commit 49fb2ce pushed by danielementary	master	24 days ago	2m 8s
✓ **.NET Core** .NET Core #174: by danielementary	master	26 days ago	3m 0s
✓ **.NET Core** .NET Core #172: by danielementary	master	26 days ago	2m 35s
✓ **Merge pull request #74 from Progin-SA-Metal/mainp...** .NET Core #170: Commit be07016 pushed by danielementary	master	last month	2m 17s
✓ **.NET Core** .NET Core #168: by danielementary	master	last month	2m 17s



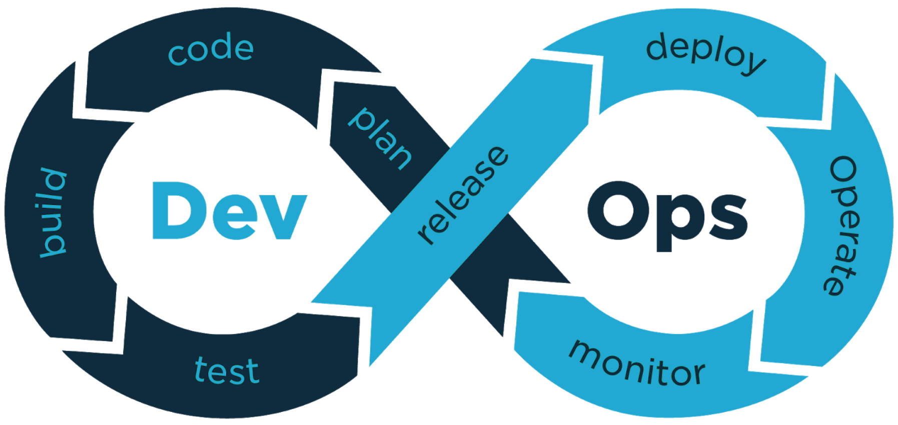
Jenkins / Blue Ocean #423
Branch master
Commit #601366d
Changes by Michael Neale, Ben Waldo and Ivan Meredith
3 minutes and 42 seconds
14 minutes ago
Pipeline
Changes
Tests
Artifacts
Re-run
Build
Test
Browser Tests
Dev
Staging
Production
JUnit
DBUnit
Jasmine
Firefox
Edge
Safari
Chrome
Edge
Start Docker container 3 minutes and 42 seconds
Warm maven caches 5 seconds
Install Java Tools 8 seconds
Maven 6 minutes and 12 seconds
Show complete log

```
[INFO] ------------------------------------------------------------------------
[INFO] Building services 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ services ---
```



The DevOps pipeline

A team using Continuous Integration with their project will typically follow a work pipeline to keep the software development process organized and focused. The DevOps pipeline describes a set of practices that the development (Dev) and operations (Ops) teams implement to build, test, and deploy their software faster and more easily. This pipeline is often divided into 8 steps as follows, and you are likely to encounter some variant of the following diagram in DevOps-related materials:



This shows how the DevOps pipeline is a cyclic process. This means that the workload is divided into multiple chunks and that the whole pipeline will be repeated multiple times.

1. **Plan.** The planning stage involves planning out the entire workflow before developers start coding. In this stage, product managers and project managers play an essential role. It's their job to create a

development roadmap that will guide the whole team along the process. The work is broken down into a list of tasks which are then assigned to team members.

2. **Develop.** In the Development stage, the developers start coding by working on the tasks that were assigned to them. They make frequent commits but only push their code once they have completed it and thoroughly tested it.

3. **Build.** When the developers push their code, the CI will automatically trigger a build. If the build fails, the developers need to react and find what makes the build work on their own machine but fail in the CI environment.

4. **Test.** If the build is successful, the CI will launch all tests ever written for the software to check that the new build hasn't broken any of its functionality.

   If all tests are successful, the new code is close to being ready for release. The developer can issue a pull request to merge the master with their own branch. This again will automatically trigger a build and test to ensure that the merging process hasn't created new bugs

5. **Release.** At this stage the code is ready to be integrated in the main code base and the pull request can be safely merged with the master branch.

6. **Deploy.** The software is now ready to be pushed to production. Different automatic deployment methods have been created for that purpose. You'll learn more about deploying your product later on in this lecture.

7. **Operate.** The new release is now live and being used by the customers.The operations team is now hard at work, making sure that everything is running smoothly. The users also have a way to submit their feedback which is recorded by the operations team

8. **Monitor.** The "final" phase of the DevOps cycle is to monitor the environment. This builds on the customer feedback provided in the Operate phase by collecting data and providing analytics on customer behaviour, performance, errors and more. The findings of the monitor phase will directly influence the plan stage for the following pipeline cycle and will help decide what tasks lie ahead.

Bells and whistles

Finding bugs using static analysis

○ Despite testing being the most used method to find bugs, it is not the only one that exists. While testing (*dynamic analysis*) implies to run the program and see if the result is the one that is expected, *static analysis* is performed without executing it. The use of the analysis can go from highlighting possible coding errors (e.g., lint tools like the one proposed by Android Studio) up to formal verification, that proves mathematically properties about the program. Formal verification is the only method that can prove the absence of bugs. It can indeed prove mathematically that a program actually does what it is specified to do. The counterpart is the time needed to run such tools can quickly become unmanageable and so it can only be used on relatively small programs. It can however be used for highly important pieces of software like medical devices software, nuclear reactor management software or aviation embedded software. There are some examples of tragic events caused by a software bug:

- The Therac-25 was a computer-controlled radiotherapy machine developed in the 1980's. A bug of concurrency caused the machine to administer doses of radiation hundreds to thousands of times greater than required, leading to serious injury or death.
- More recently the Boeing 737 Max was involved in some lethal crashes. The 737 Max needs software to correct an aerodynamic issue caused by bigger engines than the 737. This problem occurs at high angles of attack (when the nose of the plane is high) and makes the nose go even higher. To correct that, Boeing decided to measure this angle and have software that makes the plane pitching down when needed. A malfunction in this software and problems with the sensors are at the origin of some crashes. (see these links to learn more:
https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer
https://www.bloomberg.com/news/articles/2020-02-06/boeing-identifies-new-software-problem-on-grounded-737-max-jet)

Using formal verification on critical pieces of software can help avoid such tragic ends.

Code reviews

When you are done coding a new feature or correcting bugs for your software in your local branch and that you have thoroughly tested it, it is time to merge your changes with the main code base. However tests are not enough to assess the quality of your code. Indeed tests cannot detect bad practices such as breaking the abstraction, encapsulation. They cannot tell either if it is easily readable and modular, or if you have just written a bunch of spaghetti code.

To solve these issues, there is no choice but to have other developers manually checking the new code and deciding if it can go through to the main code base or if it first requires modifications. This is what we call code reviews: before merging your branch to the main code base, you first need reviewers to verify it and approve it.

Hopefully, modern tools leverage this manual task for the reviewers so that they can understand the changes in a glimpse. For example GitHub provides an interface only displaying the newly added or deleted pieces of code and allows the reviewers to directly leave comments on them.

As a final note on code reviews, please note that they also serve the purpose of having multiple people on the team understand what each piece of code does. This means that when performing a code review, the reviewers should make sure to understand what every new line of code does such that if its creator leaves the project, there are still people able to maintain that part of the codebase.

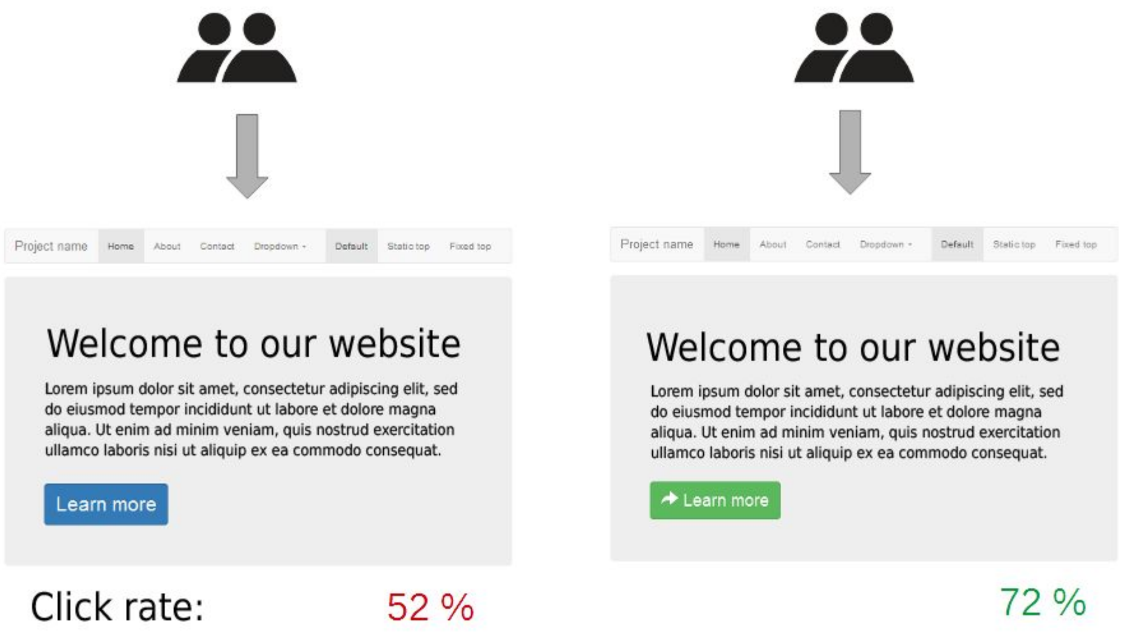
A/B testing

Knowing that the last feature you merged to the code base is thoroughly tested and is well written is a good thing, but it is also important to ask yourself if you were building the right thing. What you would think is a great addition to your software may not be perceived as such by the users. Indeed the perception of your features on your product are completely biased and is most possibly different from the one of someone using the software for the first time.

If you want to test the impact of a feature on the user, you can do a so-called A/B test. In such a test, you present half of the users with the Model A of your product, and the other half with Model B. Using different measurements, you can then assess which of Model A or Model B is the best suited.

In the example below, you present half of the people visiting your website a version with a blue button and a version with a green button. As your goal is to have the most users clicking to learn more about your product, you measure the click rate on this button. When you have collected a significant amount of data, you can in this case decide to keep the version of your website with the green button as 72% of the visitors clicked on the button vs. 52% on the other one.

In any case, one should always be careful of the common pitfalls of statistics when analyzing the results of A/B testing (small sample size, biased distribution of Model A and B, etc…)


Project name
Home
About
Contact
Dropdown
Default
Static top
Fixed top
Welcome to our website
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Learn more
Click rate:
52 %
72 %

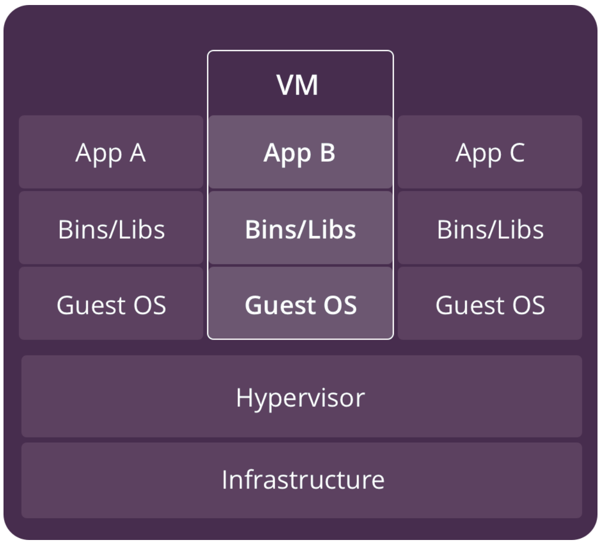
Deploying code

Virtualization and containers

We have seen that virtualization is useful in continuous integration systems, but it is also extensively used in production setups: it is core to the flexibility and scalability of modern cloud infrastructure, massive data centers and distributed computing clusters.

A virtual machine (VM) is a virtual computer with an operating system and software stack, which can live alongside other VMs on the same physical machine but is completely isolated: VMs are not aware of each other, unless explicitly specified. As such, one can run several completely different computer systems on a single machine, which also costs significantly less than maintaining multiple dedicated servers. Popular VM systems include VirtualBox, VMWare, Parallels, Oracle VM among others.

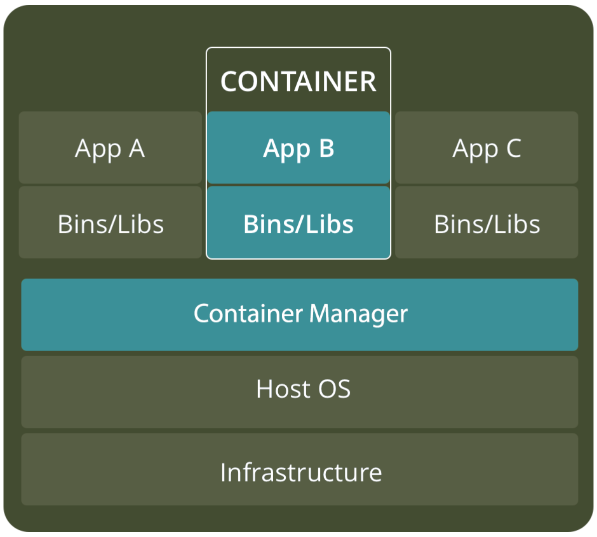
VMs allow a limited set of machines to efficiently distribute varying workloads on effective physical resources, while keeping the benefits of isolation: security (as software running on VMs cannot corrupt programs on a different VM), fault tolerance (as a process or OS crash in a VM does not impact other VMs), fast time to recovery (as a failing virtual machine can simply be restarted without shutting down hardware) and convenient maintenance (as backups simply need to store VM state changes incrementally). VMs also provide hardware independence by emulating a common baseline hardware layer at the software level, which ensures that any VM of a given virtualization system may run on any physical machine that is able to run the host virtualization software.



https://www.backblaze.com/blog/vm-vs-containers/

However, VMs inflict a non-negligible performance cost: the emulation layer consumes precious CPU cycles, and a software manager (called hypervisor) must also allocate the hardware resources to each isolated VM. Since each VM must run its own operating system, the cumulated overhead can also become quite high. As such, modern processors may not handle more than a few dozen concurrent VMs.

A more lightweight and scalable approach to virtualization is containers. Unlike VMs which virtualize the underlying hardware, container systems only virtualize the OS, i.e. the host OS kernel is usually shared, as well as its binaries and libraries in a read-only fashion. Each container can then load and execute its own runtime code and libraries, which are much smaller than a full-blown OS. The host OS implements isolation of users and programs itself through groups (a collection of processes that share a slice of varying size of resource usage, but that are completely isolated from other groups) and namespaces (the underlying kernel implementation which identifies each group). From the point of view of the processes, a container behaves similarly to a VM as the process believes it runs on its own OS and is not aware of other containers, however the performance footprint is much smaller: current processors can easily run hundreds to thousands of containers simultaneously.

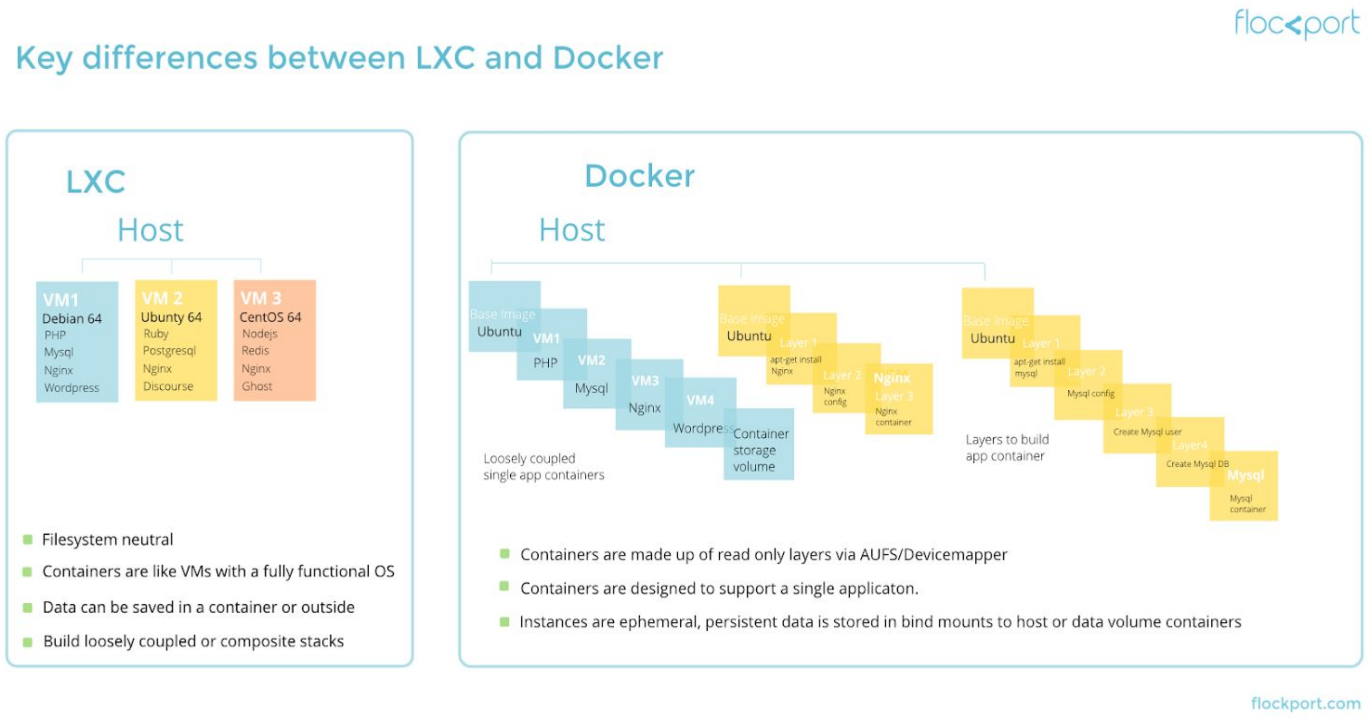


https://www.backblaze.com/blog/vm-vs-containers/

While VMs rely on hardware support to run the hypervisor with higher privileges than the VM OSes, container systems depend on the host OS kernel to implement such virtualization features, and Linux Containers (LXC) is a popular implementation for containers (side note: Windows Server also implements 2 kinds of containers, but the first (WSC) does not offer similar isolation guarantees, while the other (Hyper-V containers) is more akin to a minimal virtual machine, more details here).

Docker

Docker is a virtualization engine application that restricts the notion of containers to single application systems. Initially based on LXC, it now implements its own virtualization environment (libcontainer) which takes advantage of many hardware and OS virtualization capabilities. The main idea behind Docker is that each application lives in its own very lightweight container, and can only access other containers and the host system through user-defined communication channels. For instance, a web service may require an HTTP server (nginx), a back-end interpreter (php), a database (MySQL), a web application (Wordpress) and some storage for files; with Docker, one would download and deploy each image individually, and establish network bindings between each of them, as well as file system access to a dedicated container storage volume.



- Filesystem neutral
- Containers are like VMs with a fully functional OS
- Data can be saved in a container or outside
- Build loosely coupled or composite stacks

- Containers are made up of read only layers via AUFS/Devicemapper
- Containers are designed to support a single applicaton.
- Instances are ephemeral, persistent data is stored in bind mounts to host or data volume containers

https://archives.flockport.com/lxc-vs-docker/

Once upon a time, production software had to be deployed on dedicated machines: however, the management of the hardware, the operating system and the application stack was no easy task, as it formed an entangled system which had to be simultaneously and manually installed, configured, updated and secured (side note: this deployment model is still used in small-ish scale systems, for instance mutualized web hosting). Proprietary software is distributed through packaged binaries and open-source software can be built from source, but chances of either of them working out-of-the-box are very low; environment issues prevent it from predictably running the same way on systems with different conditions: varying hardware manufacturers, OS versions, availability of libraries (dependencies must be either bundled resulting in huge binaries sizes, or DevOps engineers need to manually ensure that the packages are installed and available on each physical machine). Additional deployment and runtime issues, such as configuration, interprocess communication channels and allocation of hardware resources can quickly become a nightmare to manage. Moving, updating and securing systems with multiple software programs can also be challenging.

Virtual machines solved parts of the problem by decoupling the software from the underlying system, but they were heavy and slow. Container engines, such as Docker, provide additional encapsulation: they wrap an application and its dependencies such that its deployment and maintenance are externalized, and like VMs they are isolated from the underlying OS and other software, without the hassle of per-application OSes.

In Docker, each application system is an image that can be built from a repository or pulled from a registry (like DockerHub). Images are immutable once published, thus managing versions is convenient, as one can always fall back to an older image if needed. Images can run as copies on multiple containers independently, and Docker provides virtualization capabilities to emulate a network or a filesystem between containers and with the host system. Docker Compose is a tool to describe complete systems by simply enumerating image names and their versions, and their virtual links in a single configuration file. Advantages of Docker include ease of installation, predictability of container state, composability of containers into complex systems, isolation, portability and scalability with a very low performance overhead.

Cloud providers and frameworks

When the computing system must scale to accommodate millions of users across the entire world, it cannot live as a single monolithic component: otherwise, a crash from a single operation would disrupt the service to all its users, which is not acceptable. Also, such a program would not be able to scale out of a single machine. One great idea would be to replicate the app process: even though sharing state across instances is not trivial, it now allows the application to live in datacenters in distributed locations, but a crash can still occur at any level in a copy. (This deployment model is still used for some small to medium scale applications, as they may not require more resilience!).

However, if the application is broken down into smaller independent containers which each handle a very specific task (e.g. only displaying a front-end web app, logging events, serving a REST API, ...), then a crash in one part of the system does not affect the others. This principle is the core idea behind the microservices architecture, in which programs are built from the ground-up to serve a single, limited purpose. Complex software systems emerge by combining microservices through network messaging.

Now, each module of the system becomes a separate program (and container) by itself, which can be replicated on-demand to scale with resource usage, restarted if it fails, spawned multiple times (in several locations in the world) and users can be split across the containers depending on their regions (i.e. load balancing). However, managing containers at scale also becomes a laborious task: scheduling them, configuring their storage volumes and setting up their networks, monitoring their health... did we get back to square one where DevOps engineers had to manually configure their infrastructure ?

Fortunately, giants such as Google who start more than 2 billion containers every week created tools to manage gigantic container systems: Kubernetes is an orchestration and automation tool for containerized workloads and services which can be summarized as a "framework to run resilient distributed systems". Its features include:

- Service discovery and load balancing: distribute traffic across containers automatically, provide internal DNS and IP systems to name and address containers, links are dynamically resolved
- Storage orchestration: mount and manage heterogeneous file systems (physical hard drives, virtual volumes, cloud storage such as AWS, data lakes, ...)
- Automated rollouts and rollbacks: update software across the entire world by replacing containers at a controlled rate, revert to previous container image if they fail or crash
- Automatic bin packing: developers only provide Kubernetes with a cluster of physical nodes, the desired CPU / RAM per container and Kubernetes will automatically fit containers on nodes for best resource usage
- Self-healing: replace failing containers, kill unresponsive containers, boot containers and Kubernetes will make them available once ready
- Secret and configuration management: passwords, tokens, keys stored in-memory in required nodes only, deleted everywhere if no node needs them anymore

The key idea behind Kubernetes is to group containers into pods (each pod shares its resources among its containers). Each pod gets its own IP address. Pods can be managed through ReplicaSets, which ensure that multiple copies of a given pod exist such that it has at least one replica running at any given time. Example users of Kubernetes include Booking.com, IBM, Spotify, Wikimedia, ...

OpenStack is an infrastructure management system to scale even beyond containerization. It provides unified management of various hardware pools for compute, storage and networking in data centers. It includes many components to use and serve pools: identity management, persistence, control through Web UI, CLI or REST API, database storage, telemetry, map-reduce compute systems, virtualization and bare metal allocation, messaging, search, container orchestration (including Kubernetes). Example users include Blizzard Entertainment, CERN, BBC, eBay, Sony (for its PS4 online infrastructure), ...

Open-source software (OSS)

Today, a lot of software and source code is distributed freely using open source licenses. A lot of important building blocks of software systems are distributed this way: frameworks (e.g., Angular, React, Rails, Spring, QT), compilers (e.g., GCC, LLVM), operating systems (e.g., GNU/Linux, Android), database management software (e.g., MySQL, mongoDB, PostgreSQL), and a lot more. This makes the lives of software engineers easier, as they can have at their fingertips multiple free and open implementations to use for building their software.

These building blocks are shared with the community for free, with a license that usually grants you four basic rights: to execute the software as you wish, to understand how it works (access the source), to redistribute it to others, and to redistribute modified copies of it. Some licenses may grant you more rights, others may grant you fewer, but the main idea remains the same: the source code is always provided, and everyone is welcome to contribute.

A lot of tech companies contribute to OSS projects. For example, Google, Apple, Facebook, Amazon, and Microsoft all maintain open-source projects that have millions of users. As they rely on open source themselves, this is a way to give back to the community. It also enables the community to make contributions to improve the product. For example, the Java programming language is implemented by Oracle (after having acquired Sun, the original creators of Java) and the OpenJDK project, Oracle providing commercial versions with extended support and OpenJDK providing the open source implementations. Other vendors, such as RedHat, help maintain the OpenJDK implementation after Oracle starts working on the next version.

Not all open-source projects are backed by a big company or non-profit. Many useful libraries are actually built by just a handful of software engineers. As a developer, you too can contribute to OSS, either by starting your own projects, or (most commonly) by helping other projects. Github has become an important hosting service for much of the open-source software out there, and the service makes it easy to find the software you want (just for fun, check out the most starred repos on Github). To contribute, you usually submit your contributions in a pull request. It is very rewarding for a developer to have a contribution accepted in a project used by thousands or even millions of users.

As the source code of OSS is, by definition, open to all to read, sometimes people modify the code to build other software. This is what we call a *fork*. In git, forking means making a whole copy of the source code into a new repository. Sometimes, you will fork just to patch a bug or add a feature. You can then submit your patch to the *upstream*, the software you originally forked. Or, usually because the patch is not accepted by the upstream, or maybe just because you want to build something different, you will maintain your fork and make it a completely new software, with its own team. This is what happened for example with LibreOffice, a fork of

OpenOffice at version 3, or with MariaDB, a fork of MySQL.

However, open source doesn't protect you against bad code or bad actors. Bugs can still exist, and, more dangerously, a project that is entirely trustworthy at some point in time may become abandoned, or even malicious a few years after.

The event-stream vulnerability involved a legitimate and popular package on NPM, a centralized package registry for open-source Javascript software. The event-stream package was downloaded about 2 million times per week at the time, but hadn't received any update for about a year. A new ill-intentioned maintainer managed to inject a malicious rewrite of the flatmap-stream package by posing as a new contributor of the library and adding the vulnerability to event-stream as a dependency. The code targeted another library called copay-dash, a bitcoin wallet platform which itself depends on event-stream, by stealing and uploading credentials to the attacker's server. Even if the code only targeted a specific usage of the library, millions of users were potentially at risk while version 3.3.6 was released in the wild, even though all of the code was open-source.

Another example is OpenSSL between 2012 and 2014. OpenSSL is an open-source software library that implements the SSL and TLS protocols and thus provides secure communication over the network. This software is written in C, which gives a lot of power to programmers, including to write code that has plenty of memory safety bugs. In 2014, it was publicly disclosed that such a bug was present in the 1.01 version of OpenSSL which had been out since 2012. In its implementation of the TLS Heartbeat extension, the developers forgot to do a proper boundary check on packets they received from the network. This allowed attackers to craft their own packets that would trigger a buffer over-read. Using this vulnerability, the attackers were able to read data directly from the victim's memory. By doing so, they could access sensitive data such as private keys. This resulted in major security issues as the attackers were now able to decode all messages encoded with the stolen private keys or to launch a man-in-the-middle attack. When this security flaw was finally disclosed, it is estimated that half a million of secure web servers certified by trusted authorities were vulnerable to this attack.