WhiteSource

THE MAIN PILLARS OF THE DevOps TOOLCHAIN

White Paper >>

THE DEAL WITH DEVOPS

Software companies often have a problem closing the gap between what the customer orders and what the engineers deliver. Usually, the main cause of this difficulty is the separation of the development environment and the production environment. After all, when an engineer only has access to the development environment, they will focus on delivering results there.

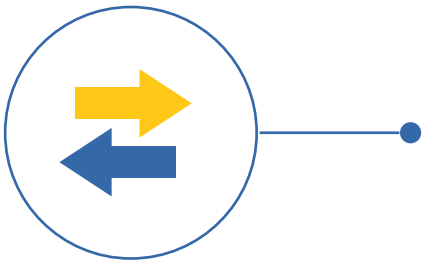
In the past, engineers produced code that was later shaped into products. Integration took care of separate units of software to make it one, coherent whole, testing provided necessary QA, and operations deployed the integrated code into production. Performing these tasks manually can take anywhere from hours to weeks. A bug in a change introduced by development could manifest itself in production much later, when it was almost forgotten by its author.

Debugging was thus costly and sometimes ineffective when development and production environments differed significantly.

Nowadays, with automation and codification of parts of a product that are not limited to the application itself (e.g., configuration, infrastructure, end-to-end tests, etc.), the gap can be made much smaller and developers can have access to a production-like environment on their own machines. This trend of bringing different parts of the release process together is known as DevOps. Even if you do not want to invest in a full continuous delivery (CD) pipeline, there are likely other DevOps tools that are relevant to your needs. In this article, we cover the most common pillars of the DevOps toolchain.

DevOps PRINCIPLES

How can you tell if a tool you are using encourages the use of DevOps principles? Although no two tools are the same, most of them share a few common qualities.



COMMUNICATION

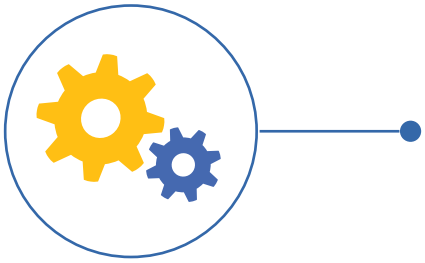
First of all, to close the gap between different stakeholders such as developers, product managers, testers, integrators, operations, and customers, you need an efficient communication method. This communication needs to be operated in terms that can be understood by all concerned parties.

DOCUMENTATION

Next, you need to maintain a thorough documentation.
With DevOps, documentation does not come in the form of a user manual.
In DevOps environments, it is preferable for the code to speak for itself.
Documentation generators can be helpful here.

Even a clearly written excerpt from a manual, like "take this Virtual Machine image and issue that command," may result in typos that lead to errors. When an instruction is reduced to run `vagrant up` and the steps themselves are written as reproducible code, mistakes are much less likely to occur. Using configuration management (CM) and Infrastructure as Code (IaC) can also serve as a good documentation.

While communication and documentation encourage knowledge sharing, the rule of least astonishment lowers the barrier of entry for newcomers. If your project follows well-known conventions like using make all to build the project and storing Python dependencies in `requirements.txt`, everyone will have an easier time navigating the project. These conventions should be consistent between your company's projects since internal migration is probably more frequent than external migration. Consistency will also make the onboarding process more straightforward.



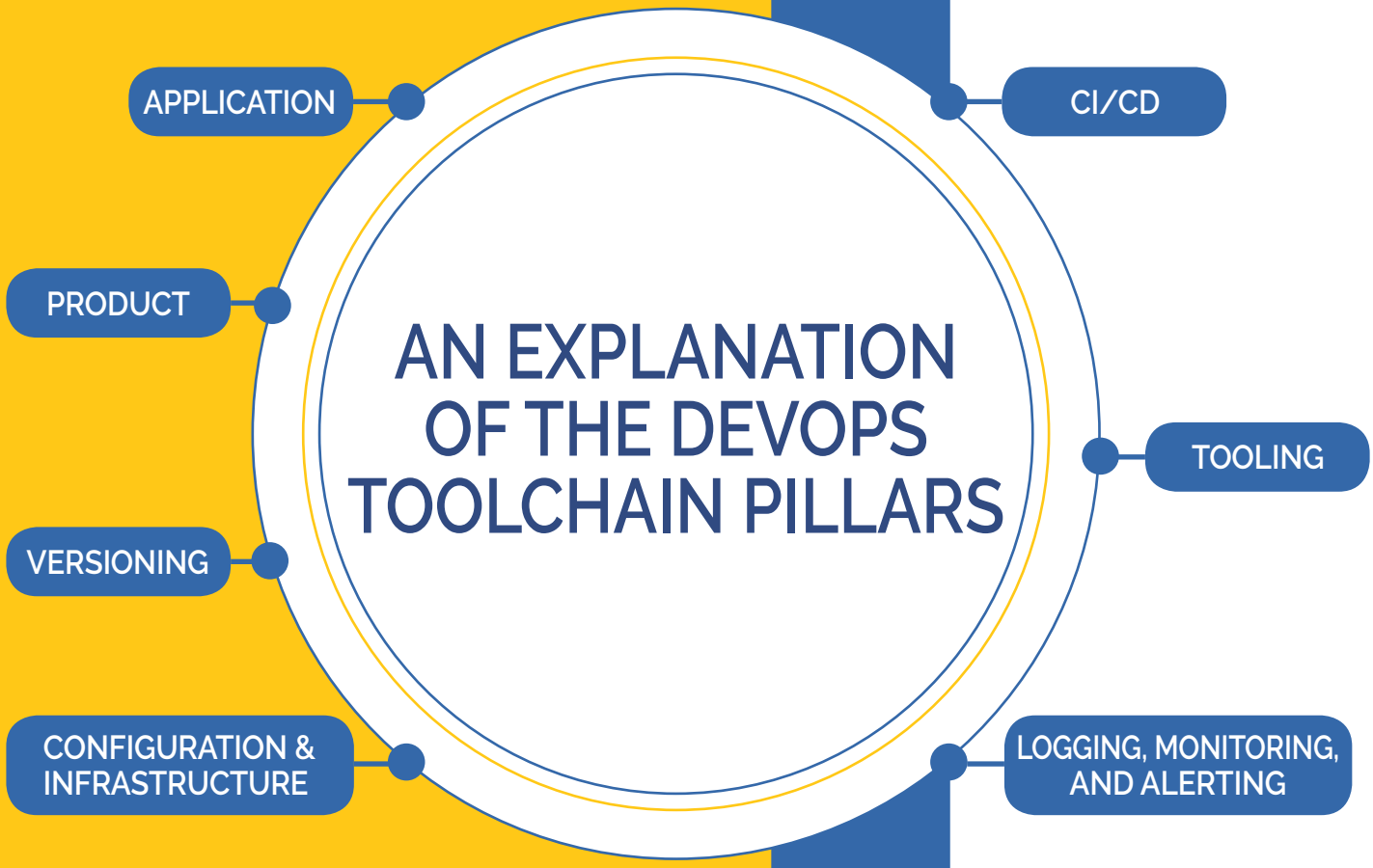
AUTOMATION

One DevOps principle that can actually be taken too far is the heavy reliance on automation. In general, the fewer manual processes there are in the product development, the better.
This concept overlaps with documentation in some ways because good automation is documentation in itself.

Sometimes, however, the cost of automation may be too high to justify even with the combined costs of manual labor and the possible need for documentation. As with most goals, you should not pursue total automation blindly. Be sure to consider the pros and cons of developing it further.

SAFETY AND SECURITY

Finally, there are the topics of safety and security that guard your product both from the outside and the inside. Hazards from the outside include various modes of hacking, attacks by botnets, and DoS. The hazard from the inside is an accidental deployment of development code to the production environment. Privilege separation and minimizing attack surfaces are some techniques that can be used to prevent hazards from both sides.



THE APPLICATION

Starting from the inside, we have the application being the heart of our product. It may be a single binary or it may consist of multiple services that are tied together. The application serves a business purpose and is the basis of a product. It is usually far from being the product itself, but nevertheless, it is vital to the product.

Even after only reading this far, you can probably tell if a project is following DevOps principles or not. In order for DevOps to work, an application needs to be modular and loosely coupled. Microservices are a great example because they need to be distributed along with a sample configuration and a means of running it as a whole. Ideally, the development environment should be as close to the production system as possible. Virtual machines and containers can be helpful here since they are able to abstract the underlying hardware to make sure it does not look different from the application's point of view. It is also beneficial to use automated tests (ideally on different levels like unit, integration, or end-to-end) and/or other functional tests like performance and penetration tests.

In short, ensure that all of your parts work as well as they need to in order to get the job done, but do not waste time perfecting miniscule details when there are more pressing changes to be made. If you think your code coverage could be better or you could optimize your code a tiny bit more, stop for a moment and ask yourself if it benefits the product both in the short- and long-run. If the answer is no, do not bother. If making the change will help with predicted growth or future debugging, it is probably worthwhile.

THE PRODUCT

A deployed and tested application as seen by a customer should be the focus of development at all times. "The App worked for me" should never be an excuse for a flawed deployment.

The product represents a successfully deployed application. With DevOps, all team members should be continuously focused on delivering the product. Just because you have committed the code and it passed the review does not mean that your work is done. DevOps says that you can consider your work done when you have prepared a code, committed the configuration, run the deployment code, and — most importantly — when the customer is able to see and interact with the product.

VERSIONING

Although almost everybody is familiar with the concept of a version control system (VCS), there are still companies using flash drives as a means of sharing source code. You can probably guess by now, but these companies are not practitioners of DevOps. Most popular, distributed VCSs allow for the main DevOps enablers: the ability to work on branches, run bisection tests to find the source of bugs or regression, and move changes between different branches easily.

Your team will use VCS very often, so it is necessary to become familiar with it. In CD systems, each push to a central repository will typically result in a deployment to some testing environment, so make sure that everyone on your team is aware of this.

One of version control's most valuable features is its ability to bring back an old, properly functioning version when you find yourself with a bug in production. Version control also has the benefit of not being limited to source code. Artifacts such as virtual machine images and container images should be versioned as well so you are always able to determine which exact elements are in the mix.

CONFIGURATION AS CODE AND INFRASTRUCTURE AS CODE

Build automation and test automation caught on quite early, but it was not until very recently that people started to care not only about the applications they produce, but also about the applications' configuration, the configuration of the system they run on, and their underlying infrastructure. Practices such as Configuration Management (CM) and Infrastructure as Code serve broader purpose than just automation.

Of course, it would be great if you could just change a few simple lines in a file and have all of the thousand servers reconfigure themselves. But, as we have said, DevOps is about more than automation — it is also about communication. Why write elaborate manuals on how to set up VMs and VPCs for your service to work if you can use a domain-specific language (DSL) to describe those steps in terms that both people and machines (like your CI/CD system) can understand? Then, you can have templates or modules of such elements ready at the beginning of your next project.

CI/CD

CI/CD is probably the term most often associated with DevOps. When you want to leverage automation, having a CI, or better yet a CD, pipeline in place helps ensure that all of the automation can actually perform in the automated environment.

CI/CD PERFORMS THE UPKEEP TASKS OF DEVELOPERS, QA ENGINEERS, RELEASE MANAGERS, ETC.

THE MOST COMMON CI/CD SCENARIO GOES LIKE THIS:

1. Receive a notification when someone commits to the repository
2. Run static analysis on that code
3. Run tests and check their status against previous runs
4. Indicate whether the change is ready to be merged into a destination branch
5. Perform a merge if requested
6. Build the project
7. Integrate with other possible projects
8. Run integration tests
9. Deploy to testing environment
10. Run smoke tests
11. Revert the deployment if tests fail

This process varies from product to product and from project to project, but the lowest common denominator is always to check the most recent change and build it. The scope of tests and analytics can differ as well, with possible steps including checking license compliance, running security checks against binary images, or running performance regression tests.

LOGGING, MONITORING, AND ALERTING

We all know that software engineers favor development over maintenance. After all, building something that works is much more fun than trying to figure out why something else breaks occasionally. We also know that few can afford the luxury of making products that will not be supported. Face it — maintenance is a cost of success. The only reason why you would not do maintenance is if you do not have any customers.

So, which DevOps practices relate to maintenance? This question goes back to the product-first mindset. You should not view maintenance as something to do only after development is finished. Instead, you should focus from day one on topics such as logging, monitoring, and alerting.

At the lowest level, your operating system, webserver, and probably a few other pieces of software you use are producing logs. To make the anticipated debugging easier, you should gather all of the information in one place for easier access.

Then, there are your applications that should produce some traces. Well-placed traces can reduce the cost of debugging by orders of magnitude. If you can spot an erratic behavior without even running a single test, you have already won half the battle.

Monitoring and alerting, on the other hand, ensure that the system is running continuously and without unexpected events. If an unexpected event occurs, monitoring and alerting inform the responsible parties right away (e.g., when disk usage is at 80%), which means they can be made aware of the event before they receive a customer report. This is also more efficient since customer reports may not help you find the source of the problem. For example, being unable to render a PDF file can have dozens of possible causes, and if the original problem was lack of disk space, that might not be the first place you would look independently.

Finally, logs, traces, and alerts provide a great basis for system auditing. You can spot security incidents, performance bottlenecks, and scenarios you have not even thought about before just by analyzing the data generated by your product. It may even help you boost sales if, for instance, you notice that customers are struggling at checkout because an application occasionally throws an error during payment processing.

TOOLING

Another important DevOps pillar is the use of appropriate tooling. This is another application of the cult of automation.

Similar people often have similar problems. When one of your engineers struggles with creating VMs necessary for testing, it means other engineers are probably having a hard time, too. So, when one person introduces a tool to ease the process in question, everyone should benefit from it, right? The answer is "yes" only if everybody actually shares the tools and does not just build for themselves. For this reason, every company that wants to encourage the free flow of ideas for improvements should offer its employees a tooling repository where they can showcase their own little helpers.

In addition to home-grown tools, it is often beneficial to use third-party tools. Why bother reinventing the wheel if your problem has already been addressed by somebody else?

What is important is that your people use the best tools available for the job. Do not force them to use hammers when the task calls for screwdrivers. Also, be open-minded about their needs, since engineers probably understand their needs better than you do.

Examples of tools that adhere to DevOps principles are: static site generators, serverless frameworks, build automation, VM management automation, container solutions, Platform-as-a-Service hosting, etc. Keep in mind the basic rules of KISS (Keep It Super Simple) and YAGNI (You Ain't Gonna Need It).

If you want a landing page, do not host it as a Django app. Instead of adding more dependencies to the project, you should consider what elements can be pared down.

DevOps AS A CULTURE

As you can see, DevOps is neither a road paved in stone to follow, nor a coherent set of rules to live by. It is better described as a culture. You can tell whether a team practices DevOps or not based on their thinking in terms of product versus project and in terms of teamwork versus individual contribution.

DevOps should not be understood by any means as a process, a dedicated team, or a piece of software. It is all about how the team works together to help get the product out in a predictable manner.

DevOps encourages a broader view that widens both space (application, testing, and configuration, deployment) and time (development time, maintenance time, future audits, and possible growth). Instead of specialized entities that check if a piece of software adheres to some rules (e.g., product specification, security requirements, scalability) after it is finished, DevOps teams embrace the various aspects of product development from day one, when every breakage is relatively easy to fix.

DEVOPS BECOME A PROFESSION THAT IMPLEMENTS THE DEVOPS CULTURE USING THESE ROLES AND TOOLS IN THE ORGANIZATION LIFE CYCLE

PUTTING THE SEC IN DEVOPS

DevSecOps is working mostly in the production side, using IT security tools like SIEM, IPS, and malware detection to deal with issues as they arise in alerts. However, there is far more that they can do to have an impact on the development side, shifting left their reach to the build stage before it reaches production. By insisting that application security testing tools are implemented during the development stage of the SDLC, the DevSecOps teams can reduce the number of issues that will need to be pushed back to the developers before the release.