DevOps explained

by Jerome Kehrli

Written in January, 2017

So ... I've read a lot of things recently on DevOps, a lot of very interesting things ... and, unfortunately, some pretty stupid as well. It seems a lot of people are increasingly considering that DevOps is resumed to mastering chef, puppet or docker containers. This really bothers me. DevOps is so much more than any tool such as puppet or docker.

This could even make me angry. DevOps seems to me so important. I've spent 15 years working in the engineering business for very big institutions, mostly big financial institutions. DevOps is a very key methodology bringing principles and practices that address precisely the biggest problem, the saddest factor of failure of software development projects in such institutions : the *wall of confusion* between developers and operators.

Don't get me wrong, in most of these big institutions being still far from a large and sound adoption of an Agile Development Methodology beyond some XP practices, there are many other reasons explaining the failure or slippage of software development projects.
But the *wall of confusion* is by far, in my opinion, the most frustrating, time consuming, and, well, quite stupid, problem they are facing.

So yeah... Instead of getting angry I figured I'd rather present here in a concrete and as precise as possible article what DevOps is and what it brings. Long story short, DevOps is not a set of tools. **DevOps is a methodology** proposing a set of **principles and practices**, period. The tools, or rather the toolchain - since the collection of tools supporting these practices can be quite extended - are only intended to support the practices.
In the end, these tools don't matter. The DevOps toolchains are today very different than they were two years ago and will be very different in two years. Again, this doesn't matter. What matters is a sound understanding of the principles and practices.

Presenting a specific toolchain is not the scope of this article, I won't mention any. There are many articles out there focusing on DevOps toolchains. I want here to take a leap backwards and present the principles and practices, their fundamental purpose since, in the end, this is what seems most important to me.

DevOps is a methodology capturing the practices adopted from the very start by the web giants who had a unique opportunity as well as a strong requirement to invent new ways of working due to the very nature of their business: the need to evolve their systems at an unprecedented pace as well as extend them and their business sometimes on a daily basis.
While DevOps makes obviously a critical sense for startups, I believe that the big corporations with large and old-fashioned IT departments are actually the ones that can benefit the most from adopting these principles and practices. I will try to explain why and how in this article.

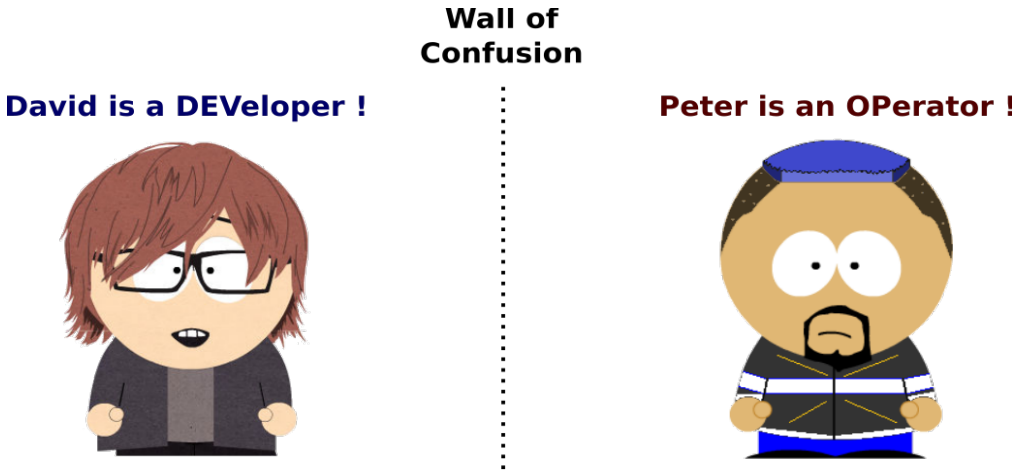
Part of this article is available as a slideshare presentation here :
https://www.slideshare.net/JrmeKehrli/devops-explained-72664261

Table of Contents

1. Introduction

DevOps is not a question of tools, or mastering chef or docker. DevOps is a methodology, a set of principles and practices that help both developers and operators reach their goals while maximizing value delivery to the customers or the users as well as the quality of these deliverables.

The problem comes from the fact that developers and operators - while both required by corporations with large IT departments - have very different objectives.



This difference of objectives between developers and operators is called the **wall of confusion**. We'll see later precisely what that means any why I consider this something big and bad.

DevOps is a methodology presenting a set of principles and practices (tools are derived from these practices) aimed at having both these personas working towards an unified and common objective : **deliver as much value as possible for the company**.

And surprisingly, for once, there is a magic silver bullet for this. Very simply, the secret is to **bring agility to the production side**!
And that, precisely that and only that, is what DevOps is about !

But there are quite a few things I need to present before we can discuss this any further.

1.1 The management credo

What is the sinews of war of IT Management ? In other words, when it comes to Software Development Projects, what does management want first and foremost ?

Any idea ?

Let me put you on tracks : what is utmost important when developing a startup ?

Improve Time To Market (TTM) of course !

The **Time To Market** or TTM is the length of time it takes from a product being conceived until its being available to users or for sale to customers. TTM is important in industries where products are outmoded quickly.
In software engineering, where approaches, business and technologies change

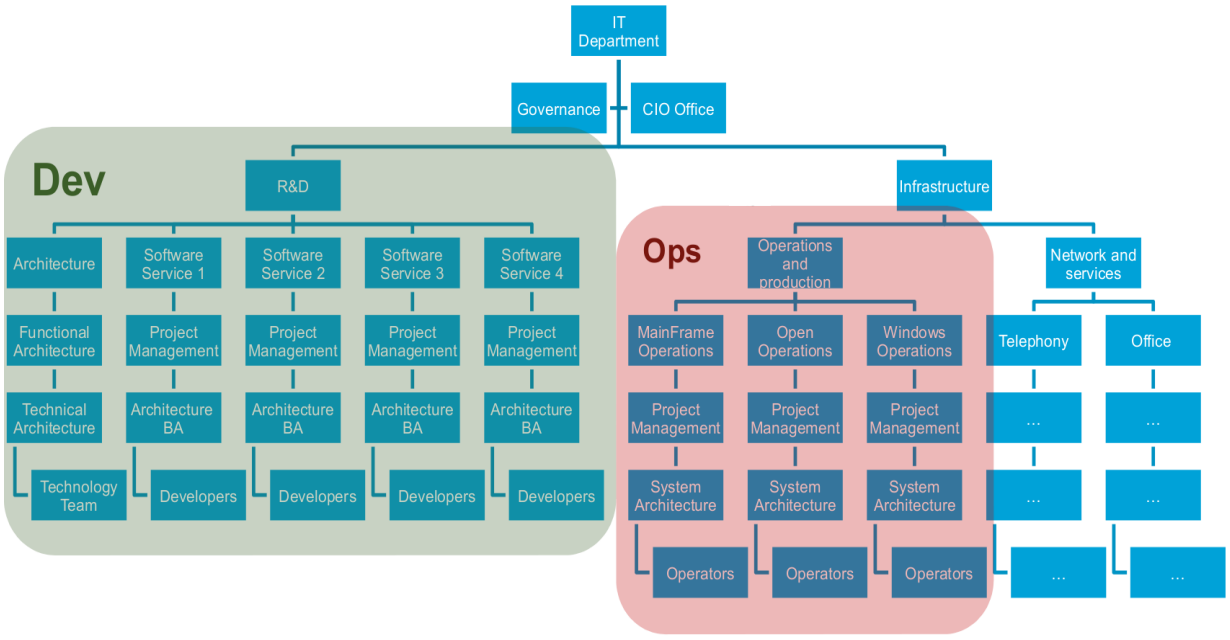
almost yearly, the TTM is a very important KPI (Key Performance Indicator).
The TTM is also very often called **Lead Time**

A first problem lays in the fact (as believed by many) that TTM and product quality are opposing attributes of a development process. As we will see below, improving quality (and hence stability) is the objective of operators while reducing lead time (and hence improving TTM) is the objective of developers.
Let me explain this.

An IT organization or department is often judged on these two key KPIs : the quality of the software, where the target is to have as little defects as possible, and the TTM, where the target is to be able to go from business ideas (often given by business users) to production - making the feature available to users or customers - as soon as possible.
The problem here is that most often these two distinct objectives are supported by two different teams : the *developers*, building the software, and the *operators*, running the software.

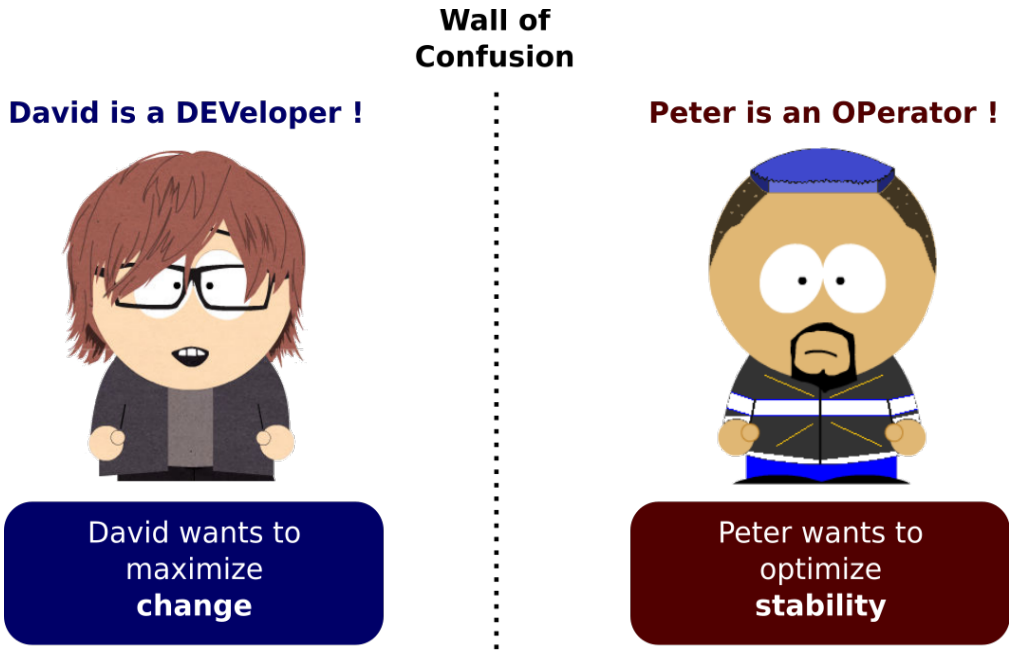
1.2 a typical IT organization

A typical IT organization, in a corporation owning an important IT department, looks as follows :



Mostly for historical reasons (operators come from the hardware and telco business most often), operators are not attached to the same branch than developers.
Developers belong to R&D while operators most of the time belong to Infrastructure department (or dedicated operation department).

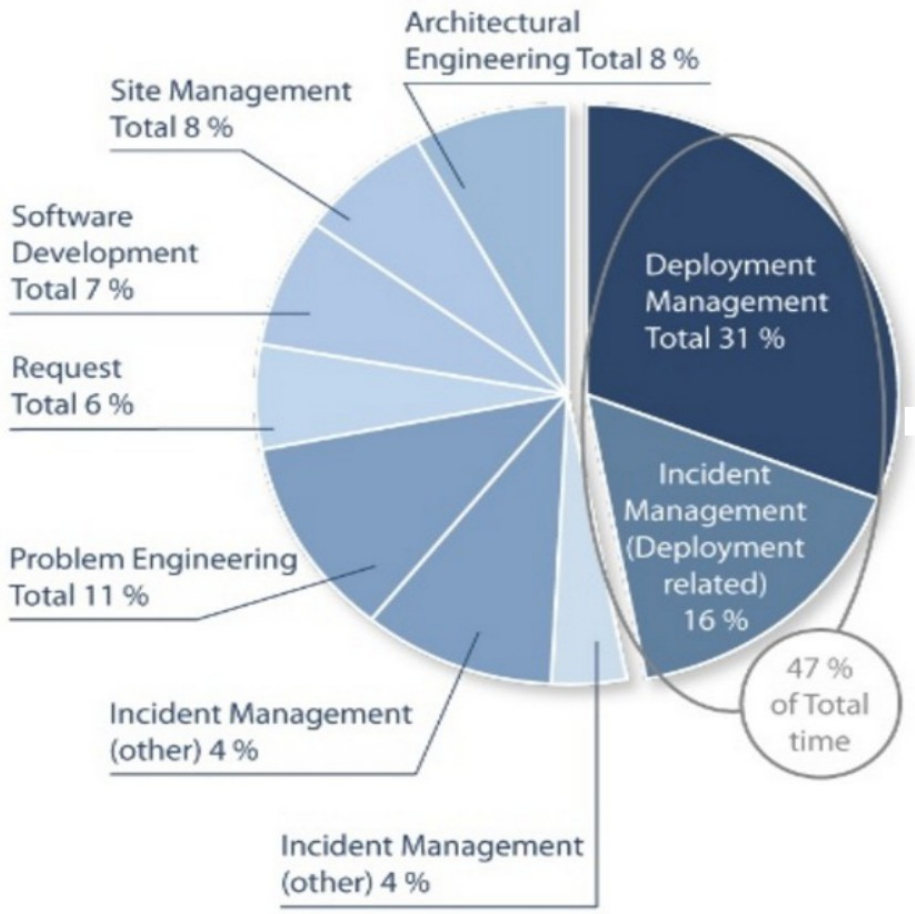
Again, they have different objectives:



In addition, and as a sidenote, these both teams sometimes even run on different budget. The development team uses the *build* budget while the operation team uses the *run* budget. These different budgets and the increasing needs to control and shorten the costs of IT in corporation tend to emphasize the opposition of objectives of the different teams.
(In parenthesis: nowadays, with the always and everywhere interconnection of people and objects pushing the digitalization of businesses and society in general, the old Plan / Build / Run framework for IT budgeting makes IMHO really no sense anymore, but that is another story)

1.3 Ops frustration

Now let's focus on operators a little and see, in average, how a typical *operation team* spends its time:



(Source : Study from Deepak Patil [Microsoft Global Foundation Services] in 2006, via James Hamilton [Amazon Web Services]
http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_POA20090226.pdf)

So almost 50% (47) of total time of Production Teams is dedicated to deployment related topics:

- Actually doing deployment or
- Fixing problems related to deployments

This is actually a pretty crazy KPI, one that should have been followed much sooner.
The truth is, operator teams have been since their inception in the early age of Computer Engineering - 40 years ago, at the time computers were massively introduced in the industry - this kind of hackers running tons of commands manually to perform their tasks. They are used to long checklists of commands or manual processes to perform their duties.
Somehow, they suffer from the "*We always did it like this*" syndrome and challenged very little their ways of working over these 40 years.
But if you think of it, this is really crazy. In average, operators spend almost 50% of their time doing deployment related tasks!

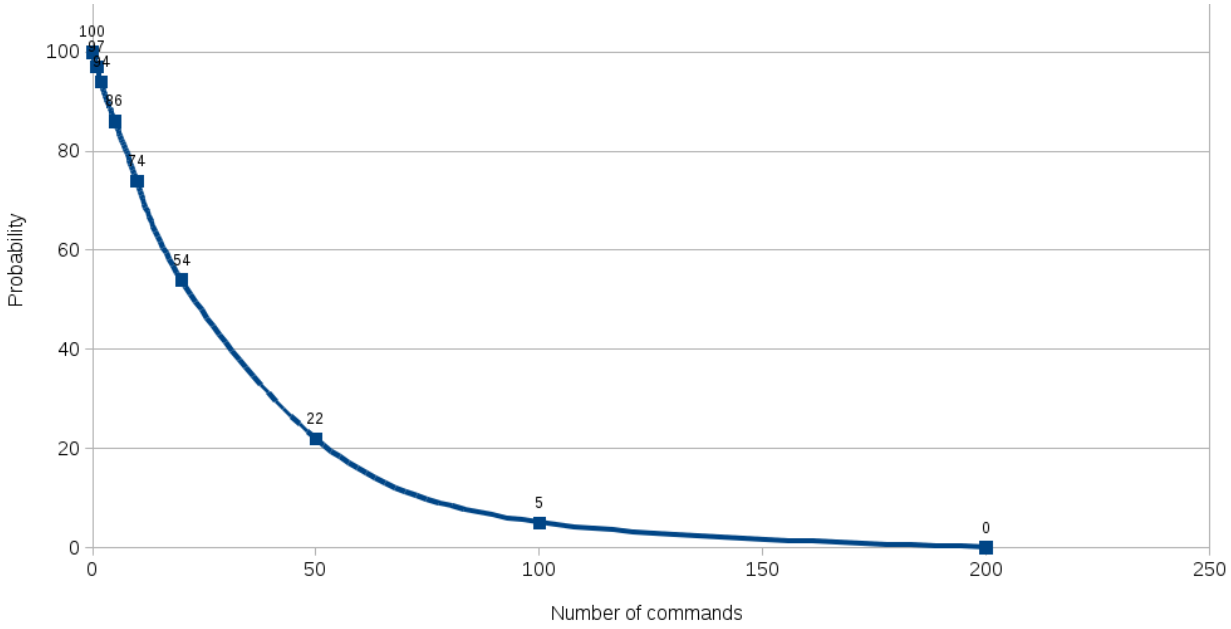
This underlines two critical needs for evoluting these processes:

1. Automate the deployments to reduce the 31% time dedicated to these currently manual tasks.
2. Industrialize them (just as software development has been industrialized, thanks to XP and Agile) to reduce the 16% related to fixing these deployment related issues.

1.4 Infrastructure automation

In this regards, another statistic is pretty enlightening:

Probability of succeeding an installation expressed as a function of the number of manual operation



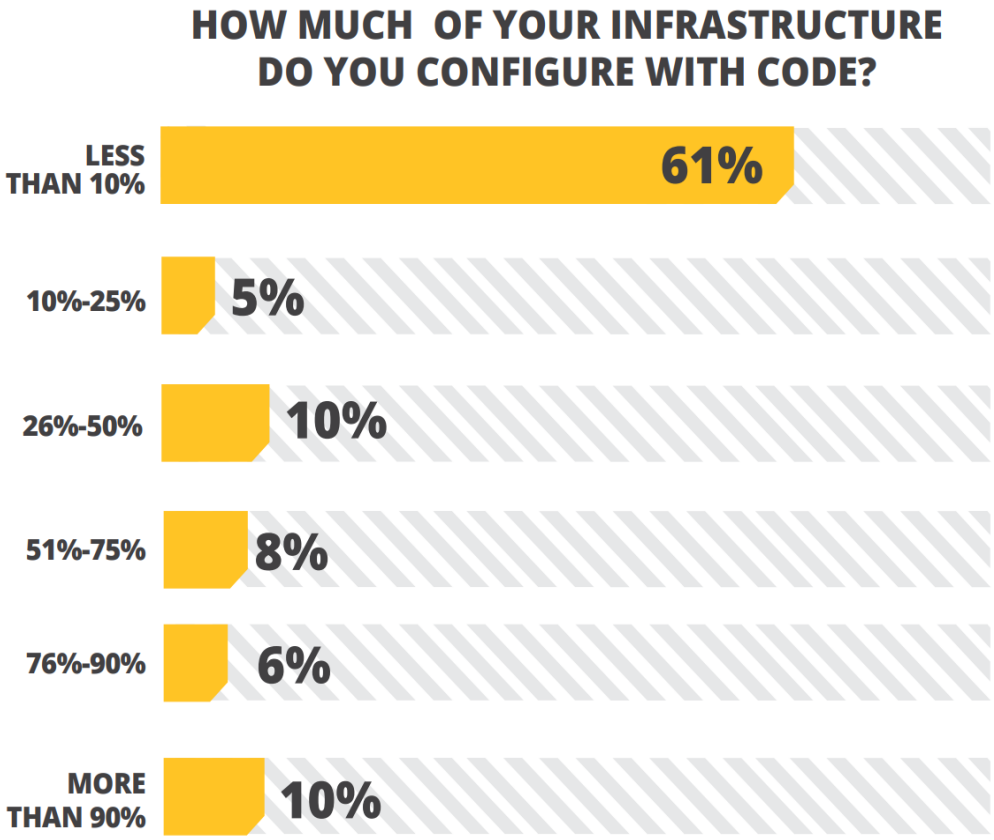
This is read the following way :

- With only 5 manual commands, the probability of succeeding an installation drops to 86% already.
- With 55 manual commands, the probability of succeeding an installation drops to 22%.
- With 100 manual commands, the probability of succeeding an installation is close to 0! (2%)%

Succeeding the installation means that the software behaves in production as intended. Failing it means something will go wrong and some analysis will be required to understand what went wrong with the installation and some patches will need to be applied or some configuration corrected.

So automating all of this and avoiding manual commands at all cost seems to be rather a good idea, doesn't it ?

So what's the status in this regards in the industry:



(Source : IT Ops & DevOps Productivity Report 2013 - Rebellabs - http://pages.zeroturnaround.com/rs/zeroturnaround/images/it-ops-devops-productivity-report-2013%20copy.pdf)

(To be perfectly honest, this statistic is pretty old - 2013 - I would expect a little different numbers nowadays)

Nonetheless, this gives a pretty good idea of how much is still to be accomplished in regards to Infrastructure automation and how much DevOps principles and practices are very important.

Again the web giants had to come up with a new approach, with new practices to address their needs of responsivness. What they started their engineering business in their early days, the practices they put in place is at the root of what is today DevOps.

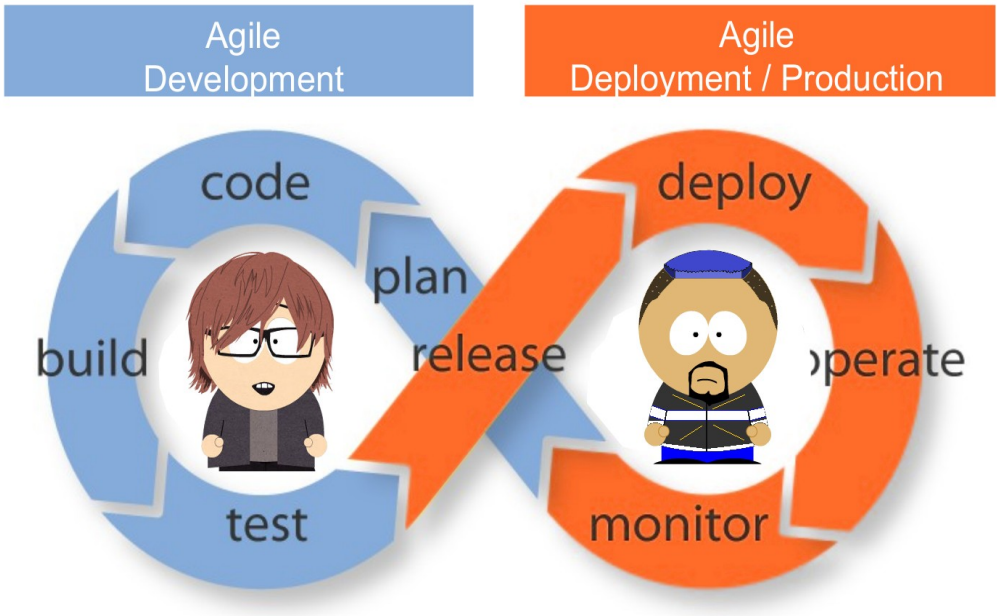
Let's look at where the web giants stand now in this regards. A few examples:

- Facebook has thousands of devs and ops, hundreds of thousands of servers. In average, an operator takes care of 500 servers (think automation is optional ?). They do two deployments a day (concept of deployment ring)
- Flickr does 10 deployments a day
- Netflix designs for failure! The software is designed from the grounds up to tolerate system failures. They test it all the time in production: 65'000 failure tests in production daily by killing random virtual machines ... and measuring that everything still behaves OK.

So what is their secret ?

1.5 DevOps : For once, a magic silver bullet

The secret is simply to **Extend Agility to Production**:



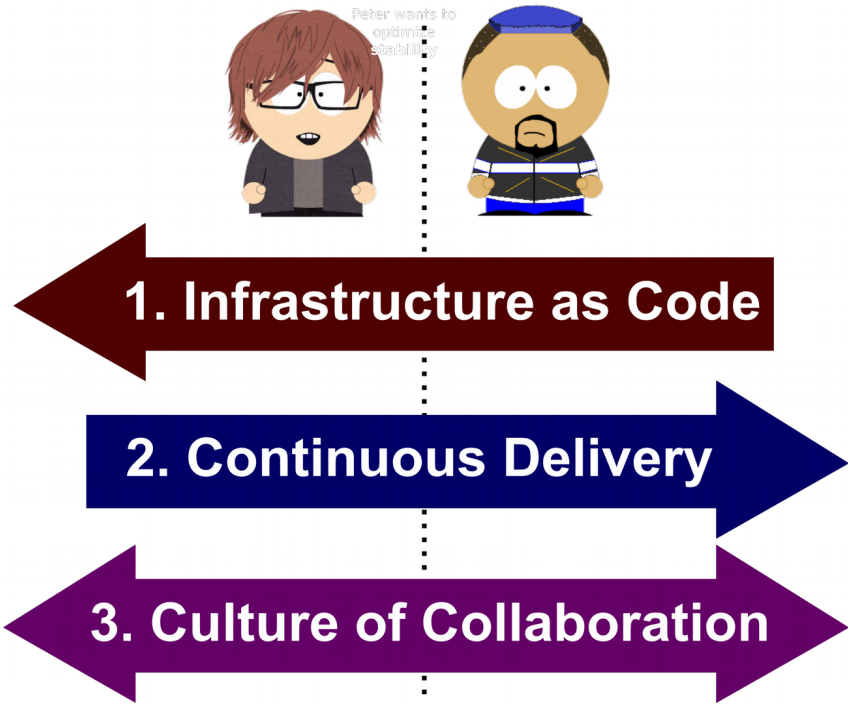
DevOps consists mostly in extending agile development practices by further streamlining the movement of software change thru the build, validate, deploy and delivery stages, while empowering cross-functional teams with full ownership of software applications - from design thru production support.

DevOps encourages **communication**, **collaboration**, **integration** and **automation** among software developers and IT operators in order to improve both the speed and quality of delivering software.

DevOps teams focus on standardizing development environments and automating delivery processes to improve delivery predictability, efficiency, security and maintainability. The DevOps ideals provide developers more control of the production environment and a better understanding of the production infrastructure.

DevOps encourages empowering teams with the autonomy to build, validate, deliver and support their own applications.

So what are the core principles ?



We'll now dig into these 3 essential principles.

2. Infrastructure as Code

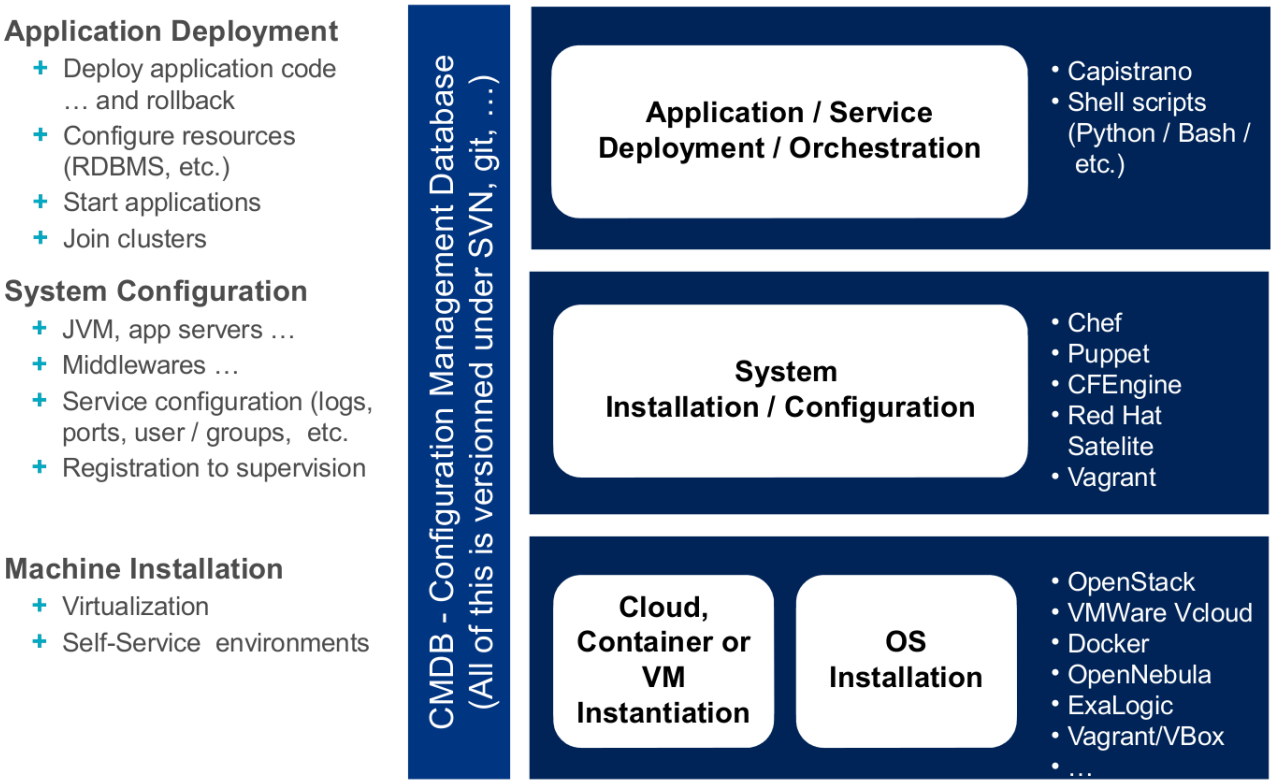
Because humans make mistakes, because the human brain is terribly bad at repetitive tasks, because humans are slow compared to a shell script, and because we are humans after all, we should consider and handle infrastructure concerns just as we handle coding concerns!

Infrastructure as code (IaC) is the prerequisite for common DevOps practices such as version control, code review, continuous integration and automated testing. It consists in **managing** and **provisioning**computing infrastructure (containers, virtual machines, physical machines, software installation, etc.) and their configuration **through machine-processable definition** files or scripts, rather than the use of interactive configuration tools and manual commands.

I cannot stress enough how much this is a key principle of DevOps. It is really applying software development practices to servers and infrastructure.
Cloud computing enables complex IT deployments modeled after traditional physical topologies. We can automate the build of complex virtual networks, storage and servers with relative ease. Every aspect of server environments, from the

infrastructure down to the operating system settings, can be codified and stored in a version control repository.

2.1 Overview



In a very summarized way, the levels of infrastructure and operation concerns at which automation should occur is represented on this schemaThe tools proposed as examples on the schema above are very much oriented towards *building* the different layers. But a devops toolchain does much more than that.
I think it's time I tell a little more about the notion of DevOps Toolchains.

2.2 DevOps Toolchains

Because DevOps is a cultural shift and collaboration between development, operations and testing, there is no single DevOps tool, rather, again, a set ogf them, or *DevOps toolchain* consisting of multiple tools. Such tools fit into one or more of these categories, which is reflective of the software development and delivery process:

- **Code** : Code development and review, version control tools, code merging
- **Build** : Continuous integration tools, build status
- **Test** : Test and results determine performance
- **Package** : Artifact repository, application pre-deployment staging

- **Release** : Change management, release approvals, release automation
- **Configure** : Infrastructure configuration and management, Infrastructure as Code tools
- **Monitor** : Applications performance monitoring, end user experience

Though there are many tools available, certain categories of them are essential in the DevOps toolchain setup for use in an organization.

Tools such as Docker (containerization), Jenkins (continuous Integration), Puppet (Infrastructure building) and Vagrant (virtualization platform) among many others are often used and frequently referenced in DevOps tooling discussions as of 2016.

Versioning, Continuous Integration and Automated testing of infrastructure components

The ability to **version** the infrastructure - or rather the infrastructure building scripts or configuration files - as well as the ability to **automated test** it are very important.
DevOps consists in finally adopting the same practices XP brought 30 years ago to software engineering to the production side.
Even further, Infrastructure elements should be **continuously integrated** just as software deliverables.

2.3 Benefits

There are so many benefits to DevOps. A non-exhaustive list could be as follows:

- **Repeatability and Reliability** : building the production machine is now simply running that script or that puppet command. With proper usage of docker containers or vagrant virtual machines, a production machine with the Operating System layer and, of course, all the software properly installed and configured can be set up by typing one single command - **One Single Command**. And of course this building script or mechanism is continuously integrated upon changes or when being developed, continuously and automatically tested, etc.
  Finally we can benefit on the operation side from the same practices we use with success on the software development side, thanks to XP or Agile.
- **Productivity** : one click deployment, one click provisioning, one click new environment creation, etc. Again, the whole production environment is set-up using one single command or one click. Now of course that command can well run for hours, but during that time the operator can focus on more interesting things, instead of waiting for a single individual command to complete before typing the next one, and that sometimes for several days...

- **Time to recovery !** : one click recovery of the production environment, period.
- **Guarantee that infrastructure is homogeneous** : completely eliminating the possibility for an operator to build an environment or install a software slightly differently every time is the only way to guarantee that the infrastructure is perfectly homogeneous and reproducible. Even further, with version control of scripts or puppet configuration files, one can rebuild the production environment precisely as it was last week, last month, or for that particular release of the software.
- **Make sure standards are respected** : infrastructure standards are not even required anymore. The standard is the code.
- **Allow developer to do lots of tasks themselves** : if developers become themselves suddenly able to re-create the production environment on their own infrastructure by one single click, they become able to do a lot of production related tasks by themselves as well, such as understanding production failures, providing proper configuration, implementing deployment scripts, etc.

These are the few benefits of IaC that I can think of by myself. I bet there are so many much more (suggestions in comments are welcome).

3. Continuous Delivery

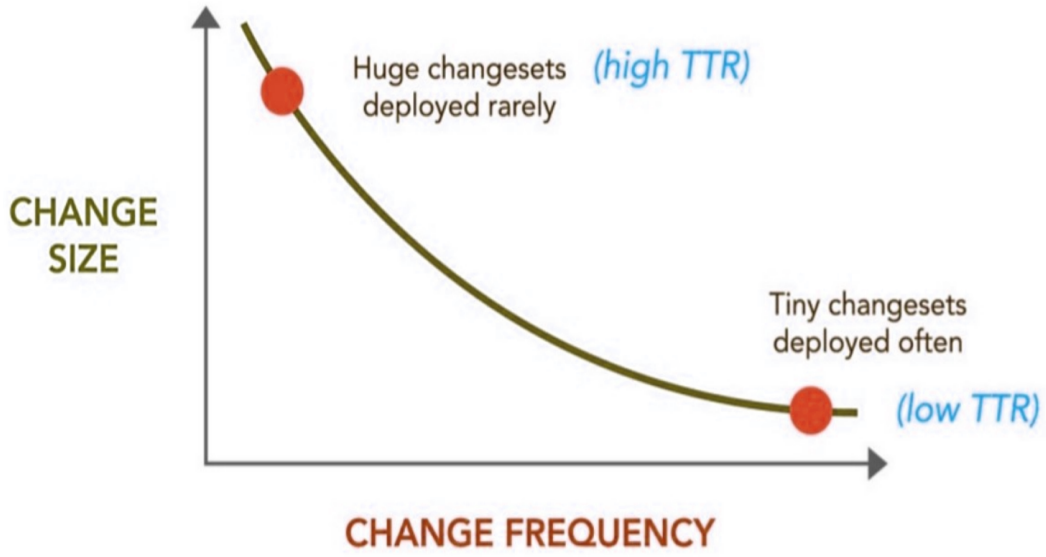
Continuous delivery is an approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently.
The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery.

Important note : Continuous Delivery ≠ Continuous Deployment - continuous delivery is sometimes confused with continuous deployment. Continuous deployment means that every change is automatically deployed to production. Continuous delivery means that the team ensures every change can be deployed to production but may choose not to do it, usually due to business reasons. In order to do continuous deployment one must be doing continuous delivery

The key ideas behind continuous deliveries are:

- **The more often you deploy, the more you master the deployment process and the better you automate it**. If you have to do something 3 times a day, you **will** make it bullet proof and reliable soon enough, when you will be fed up of fixing the same issues over and over again.

- **The more often you deploy, the smallest will be the changesets you deploy** and hence the smallest will be the risk of something going wrong, or the chances of losing control over the changesets
- **The more often you deploy, the best will be your TTR (Time to Repair / Resolution)** and hence the sooner will be the feedback you will get from your business users regarding that feature and the easier it will be to change some things here and there to make it perfectly fit their needs (TTR is very similar to TTM in this regards).



(Source : Ops Meta-Metrics: The Currency You Pay For Change - http://fr.slideshare.net/jallspaw/ops-metametrics-the-currency-you-pay-for-change-4608108)

But continuous delivery is more than building a shippable, production-ready version of the product as often as possible. Continuous delivery refers to 3 key practices:

- Learn from the fields
- Automation
- Deploy more often

3.1 Learn from the field

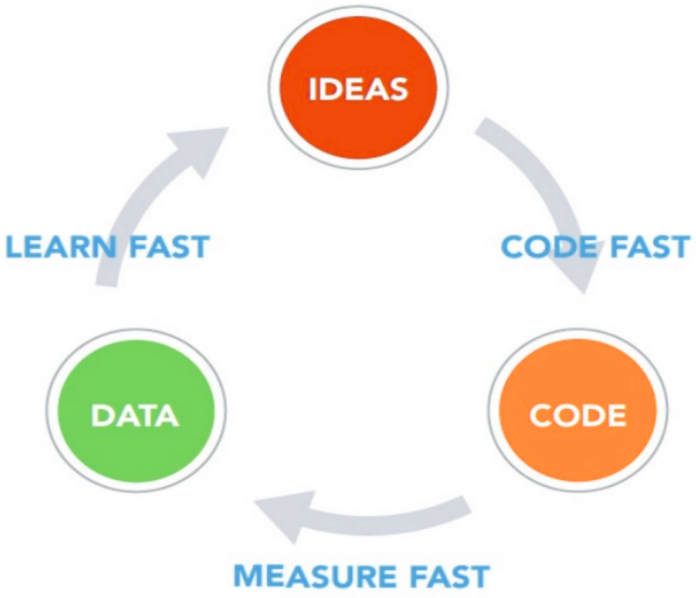
Continuous Delivery is key to be able to **learn from the field**. There is no truth in the development team, the truth lies in the head of the business users. Unfortunately, no one is able to really clearly express his mind, his will in a specification document, no matter the time he dedicates to this task. This is why Agility attempts to put the feature in the hands of the users to get their feedback as soon as possible, at all cost.

Doing Continuous delivery, as far as continuous deployment, and hence reducing lead time to its minimal possible value, is key to be able to learn the truth from the users, as soon as possible

But the truth doesn't come out in the form of a formal user feedback. One should never trust its users or rely on formal feedback to learn from users. One should trust its own measures.
Measure obsession is a very important notion from the *Lean Startup* movement but it's also very important in DevOps. One should measure everything! Finding the right metrics enabling the team to learn about the success or failures of an approach, about what would be better and what has the most success can be sometimes tricky. One should always take too many measures instead of missing the one that would enable the team to take an enlightened decision.

Don't think, know! And the only way to know is to measure, measure everything: response times, user think times, count of displays, count of API calls, click rate, etc. but not only. Find out about all the metrics that can give you additional insights about the user perception of a feature and measure them, all of them!

This can be represented as follows:



3.2 Automation

Automation has already been discussed above in section 2. Infrastructure as code.

I just want to emphasize here that continuous delivery is impossible without a properly and 100% automation of all infrastructure provisioning and deployment related tasks.
This is very important, let me repeat it once more: setting up an environment and deploying a production ready version of the software should take one click, one

command, it should be entirely automated. Without it, it's impossible to imagine deploying the software several times a day.

In section 3.5 Zero Downtime Deployments below we will mention additional important techniques helping Continuous Delivery as well.

3.3 Deploy more often

The DevOps credo is:

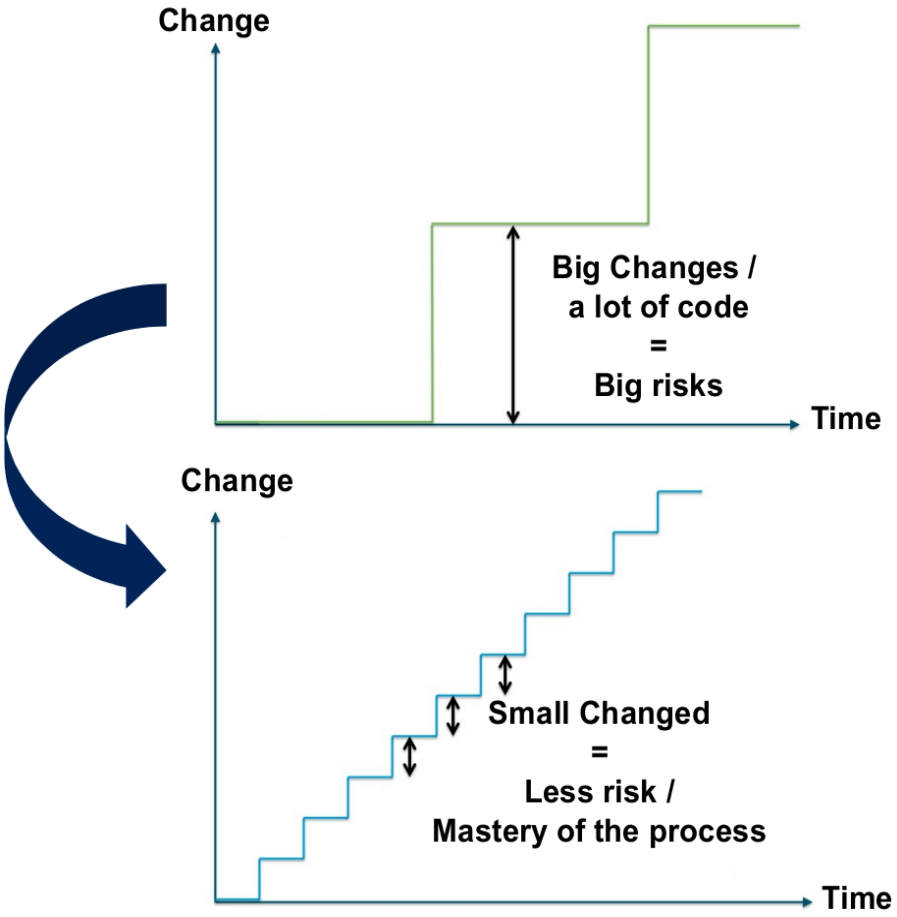
"If it hurts, do it more often !"

This idea of doing painful things more frequently is very important in agile thinking. Automated Testing, refactoring, database migration, specification with customers, planning, releasing - all sorts of activities are done as frequently as possible.

There are three good reasons for that:

1. Firstly most of these tasks become much more difficult as the amount of work to be done increases, but when broken up into smaller chunks they compose easily.
   Take Database migration for instance: specifying a large database migration involving multiple tables is hard and error prone. But if you take it one small change at a time, it becomes much easier to get each one correct.
   Furthermore you can string small migrations together easily into a sequence. Thus when one decomposes a large migration into a sequence of little ones, it all becomes much easier to handle. (As a sidenote, this is the essence of database refactoring)
2. The second reason is *Feedback*. Much of agile thinking is about setting up feedback loops so that we can learn more quickly. Feedback was already an important and explicit value of Extreme Programming. In a complex process, like software development, one has to frequently check where one stands and make course corrections. To do this, one must look for every opportunity to add feedback loops and increase the frequency with which one gets feedback so one can adjust more quickly.
3. The third reason is *practice*. With any activity, we improve as we do it more often. Practice helps to iron out the kinks in the process, and makes one more familiar with signs of something going wrong. If you reflect on what you are doing, you also come up with ways to improve your practice.
   With software development, there's also in addition the potential for automation. Once one has done something a few times, it's easier to see how to automate it, and more importantly one becomes more motivated to

automate it. Automation is especially helpful because it can increase speed and reduce the chance for error.



Now one question remains : **how often to deliver with DevOps ?**

There is no straight answer to that. It really depends on the product, the team, the market, the company, the users, the operation needs, etc.
My best answer would be as follows: If you don't deliver at least every 2 weeks - or at the end of your sprint duration period - you do not even do Agile, not to speak of DevOps.
DevOps encourages to deliver as frequently as possible. In my understanding (please challenge that in the comments if you like), you should train your team to be able to deliver as frequently as possible. A sound approach, the one I'm using with my team is to deliver twice a day on a QA environment. The delivery process is fully automated: twice a day, at noon and at midnight, the machinery starts, builds the software components, runs integration tests, builds the Virtual Machines, start them, deploys the software components, configures them, runs functional tests, etc.

3.4 Continuous Delivery requirements

What does one need **before** being able to move to Continuous Delivery?
My checklist, in a raw fashion :

- Continuous integration of both the software components development as well as the platform provisioning and setup.
- TDD - Test Driven Development. This is questionable ... But in the end let's face it: TDD is really the single and only way to have an acceptable coverage of the code and branches with unit tests (and unit tests makes is so much easier to fix issues than integration or functional tests).
- Code reviews ! At least codereviews ... pair programming would be better of course.
- Continuous auditing software - such as Sonar.
- Functional testing automation on production-level environment
- Strong non-functional testing automation (performance, availability, etc.)
- Automated packaging and deployment, independent of target environment

Plus sound software development practices when it comes to managing big features and evolutions, such as *Zero Downtime Deployments* techniques.

3.5 Zero Downtime Deployments

"Zero Downtime Deployment (ZDD) consists in deploying a new version of a system without any interruption of service."

ZDD consists in deploying an application in such a way that one introduces a new version of an application to production without making the user see that the application went down in the meantime. From the user's and the company's point of view it's the best possible scenario of deployment since new features can be introduced and bugs can be eliminated without any outage.

I'll mention 4 techniques:

1. Feature Flipping
2. Dark launch
3. Blue/Green Deployments
4. Canari release

Feature flipping

Feature flipping allows to enable / disable features while the software is running. It's really straightforward to understand and put in place: simply use a configuration properly to entirely disable a feature from production and only activate it when its completely polished and working well.

For instance to disable or activate a feature globally for a whole application:

```
if Feature.isEnabled('new_awesome_feature')
    # Do something new, cool and awesome
else
    # Do old, same as always stuff
end
```

Or if one wants to do it on a per-user basis:

```
if Feature.isEnabled('new_awesome_feature', current_user)
    # Do something new, cool and awesome
else
    # Do old, same as always stuff
end
```

Dark Launch

The idea of *Dark Launch* is to use production to simulate load!

It's difficult to simulate load of a software used by hundreds of millions of people in a testing environment.
Without realistic load tests, it's impossible to know if infrastructure will stand up to the pressure.

Instead of simulating load, why not just deploy the feature to see what happens without disrupting usability?
Facebook calls this a *dark launch* of the feature.

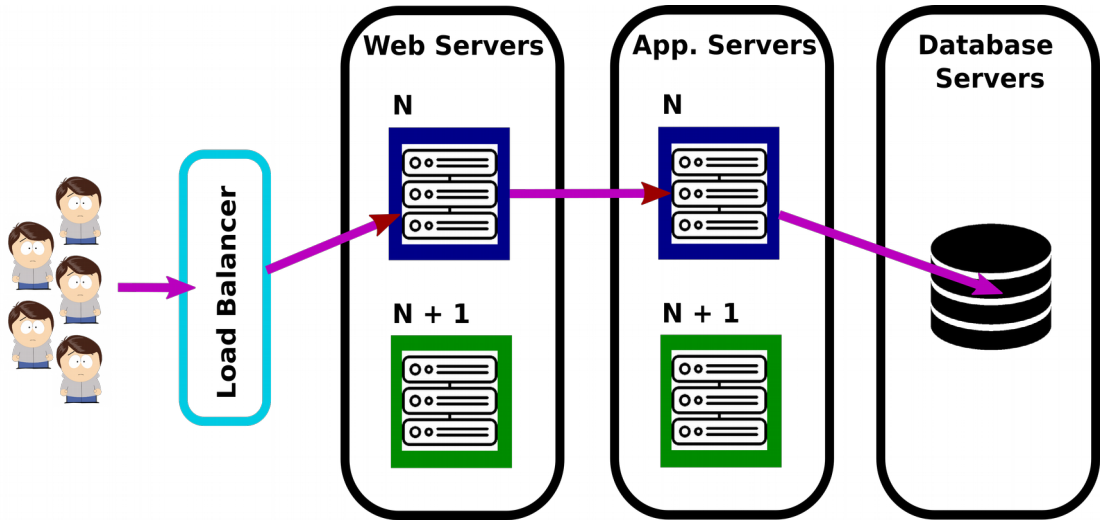
Let's say you want to turn a static search field used by 500 million people into an autocomplete field so your users don't have to wait as long for the search results. You built a web service for it and want to simulate all those people typing words at once and generating multiple requests to the web service.
The dark launch strategy is where you would augment the existing form with a hidden background process that sends the entered search keyword to the new autocomplete service multiple times.
If the web service explodes unexpectedly then no harm is done; the server errors

would just be ignored on the web page. But if it does explode then, great, you can tune and refine the service until it holds up.

There you have it, a real world load test.

Blue/Green Deployments

Blue/Green Deployments consists in building a second complete line of production for version N + 1. Both development and operation teams can peacefully build up version N + 1 on this second production line.
Whenever the version N + 1 is ready to be used, the configuration is changed on the load balancer and users are automatically and transparently redirected to the new version N + 1.
At this moment, the production line for version N is recovered and used to peacefully build version N + 2.
And so on.



(Source : Les Patterns des Géants du Web – Zero Downtime Deployment - http://blog.octo.com/zero-downtime-deployment/)

This is quite effective and easy but the problem is that it requires to double the infrastructure, amount of servers, etc.
Imagine if Facebook had to maintain a complete second set of its hundreds of thousands of servers.

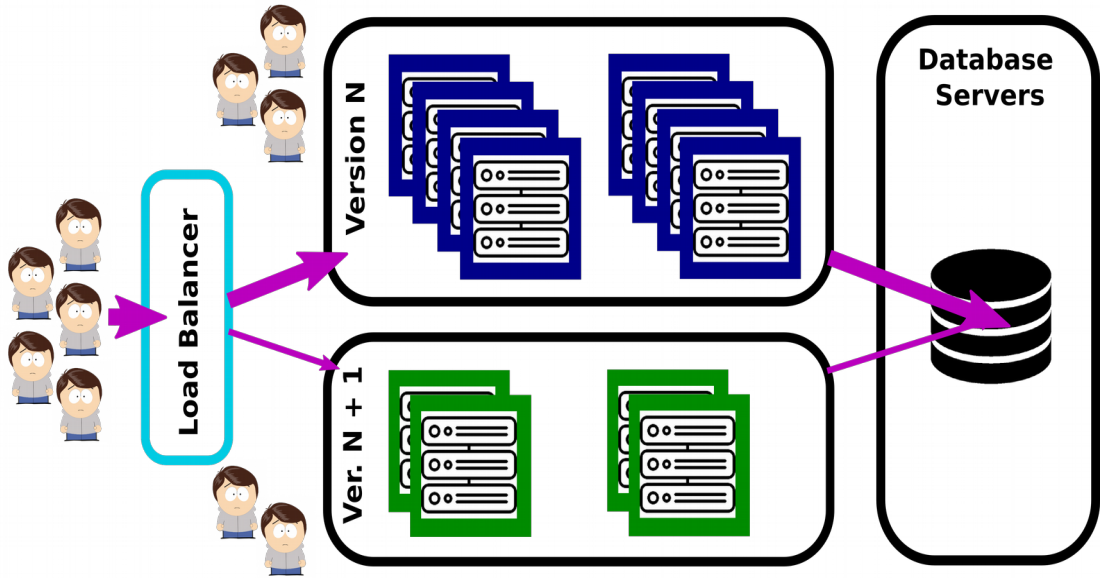
So there is some room for something better.

Canari release

Canari release is very similar in nature to *Blue/Green Deployments* but it addresses the problem to have multiple complete production lines.
The idea is to switch users to the new version in an incremental fashion : as more servers are migrated from the version N line to the version N + 1 line, an equivalent

proportion of users are migrated as well.
This way, the load on every production line matches the amount of servers.

At first, only a few servers are migrated to version N + 1 along with a small subset of the users. This also allows to test the new release without risking an impact on all users.
When all servers have eventually been migrated from line N to line N + 1, the release is finished and everything can start all over again for release N + 2.



(Source : Les Patterns des Géants du Web – Zero Downtime Deployment - http://blog.octo.com/zero-downtime-deployment/)

4. Collaboration

Agile software development has broken down some of the silos between requirements analysis, testing and development. Deployment, operations and maintenance are other activities which have suffered a similar separation from the rest of the software development process. The DevOps movement is aimed at removing these silos and encouraging collaboration between development and operations.
Even with the best tools, DevOps is just another buzzword if you don't have the right culture.

The primary characteristic of DevOps culture is increased collaboration between the roles of development and operations. There are some important cultural shifts, within teams and at an organizational level, that support this collaboration.

This addresses a very important problem that is best illustrated with the following meme:

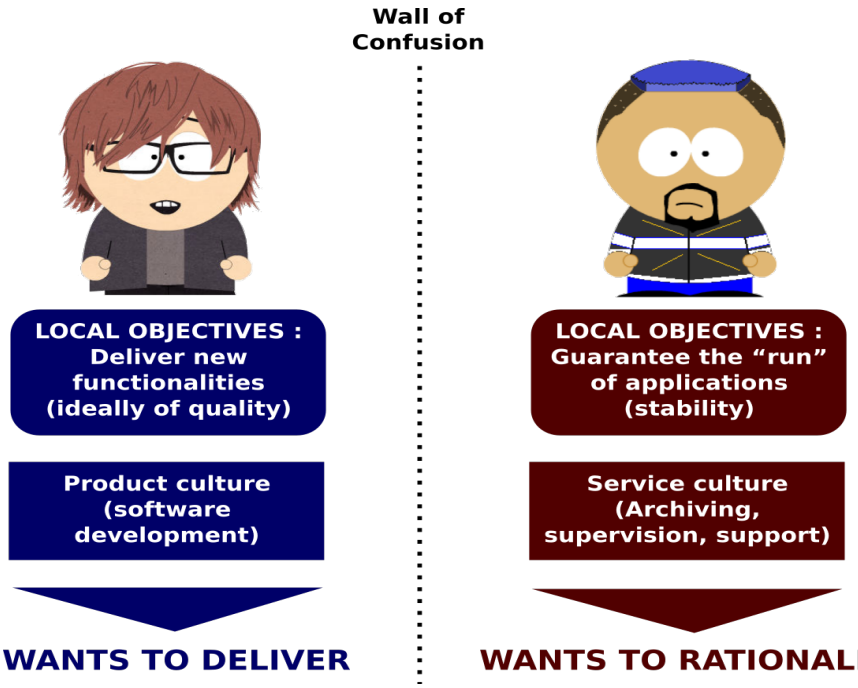
(Souce : DevOps Memes @ EMCworld 2015 - http://fr.slideshare.net/bgracely/devops-memes-emcworld-2015)

Team play is so important to DevOps that one could really sum up most of the methodology's goals for improvement with two C's: collaboration and communication. While it takes more than that to truly become a DevOps workplace, any company that has committed to those two concepts is well on its way.

But why is it so difficult ?

4.1 The wall of confusion

Because of the wall of confusion :



In a traditional development cycle, the development team kicks things off by "throwing" a software release "over the wall" to Operations.
Operations picks up the release artifacts and begins preparing for their deployment.
Operations manually hacks the deployment scripts provided by the developers or, most of the time, maintains their own scripts.
They also manually edit configuration files to reflect the production environment, which is significantly different than the Development or QA environments.
At best they are duplicating work that was already done in previous environments, at worst they are about to introduce or uncover new bugs.

The IT Operations team then embarks on what they understand to be the currently correct deployment process, which at this point is essentially being performed for the first time due to the script, configuration, process, and environment differences between Development and Operations.
Of course, somewhere along the way a problem occurs and the developers are called in to help troubleshoot. Operations claims that Development gave them faulty code. Developers respond by pointing out that it worked just fine in their environments, so it must be the case that Operations did something wrong.
Developers are having a difficult time even diagnosing the problem because the configuration, file locations, and procedure used to get into this state is different then what they expect. Time is running out on the change window and, of course, there isn't a reliable way to roll the environment back to a previously known good state.

So what should have been an eventless deployment ended up being an all-hands-on-deck fire drill where a lot of trial and error finally hacked the production environment

into a usable state.
It **always** happens this way, always.

Here comes DevOps

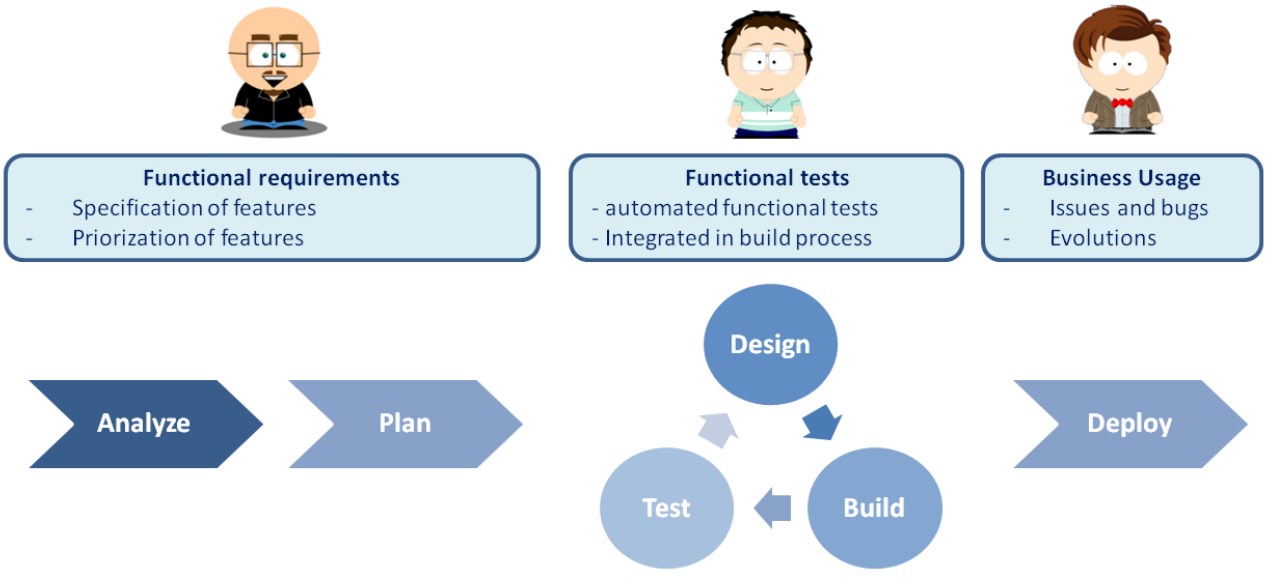
DevOps helps to enable IT alignment by aligning development and operations roles and processes in the context of shared business objectives. Both development and operations need to understand that they are part of a unified business process. DevOps thinking ensures that individual decisions and actions strive to support and improve that unified business process, regardless of organizational structure.

Even further, as Werner Vogel, CTO of Amazon, said in 2014 :

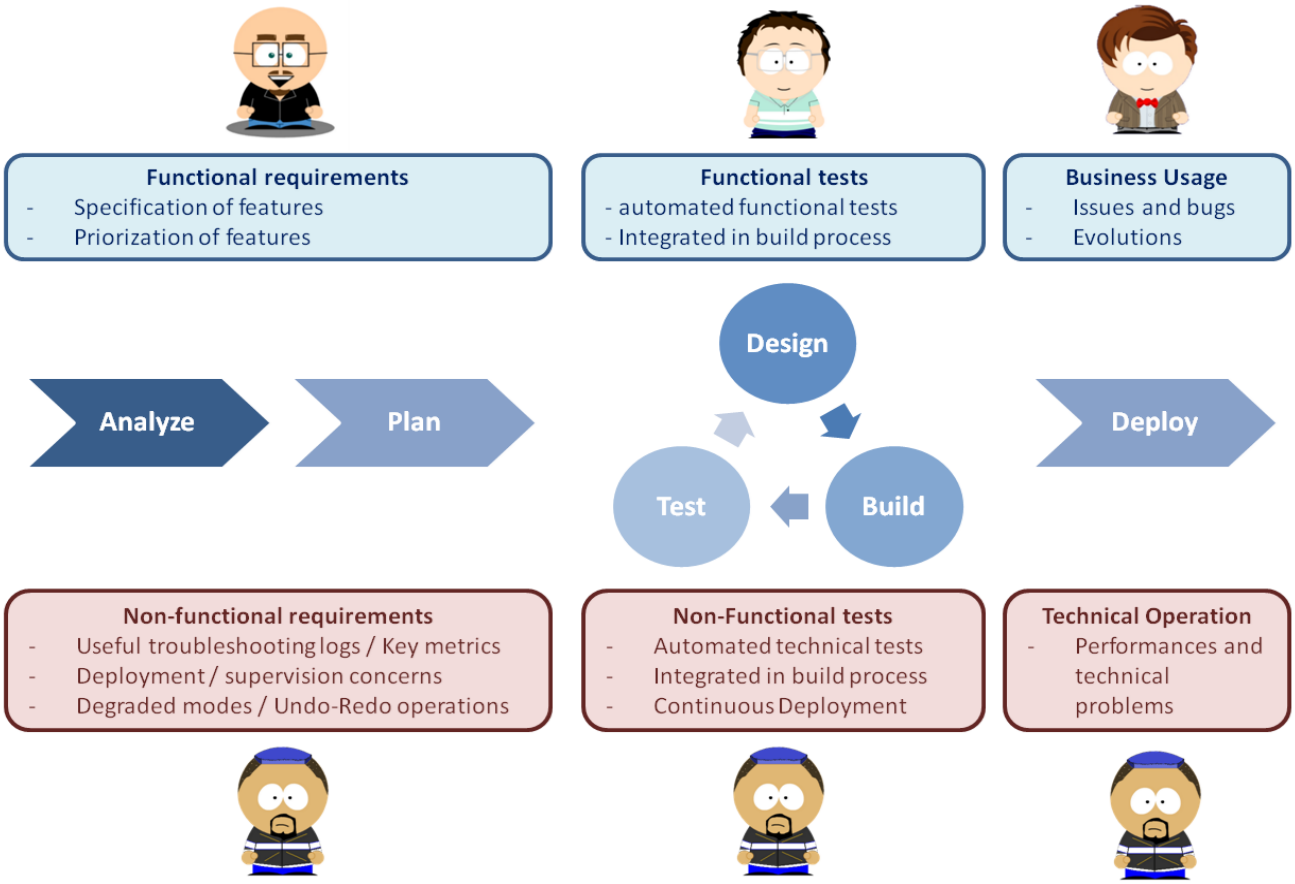
"You build it, you run it."

4.2 Software Development Process

Below is a simplified view of how the Agile Software Development Process usually looks like.
Initially the business representatives work with the Product Owner and the Architecture Team to define the software, either through Story Mapping with User stories or with more complete specification.
Then the development team develops the software in short development sprints, shipping a production ready version of the software to the business users at the end of every sprint in order to capture feedback and get directions as often and as much as possible.
Finally, after every new milestone, the software is deployed for wide usage to all business lines.



The big change introduced by DevOps is the understanding that **operators are the other users of the software !** and as such they should be fully integrated in the Software Development Process.
At specification time, operators should give their non-functional requirements just as business users give their functional requirement. Such non-functional requirements should be handled with same important and priority by the development team.
At implementation time, operators should provide feedback and non-functional tests specifications continuously just as business users provides feedback on functional features.
Finally, operators become users of the software just as business users.



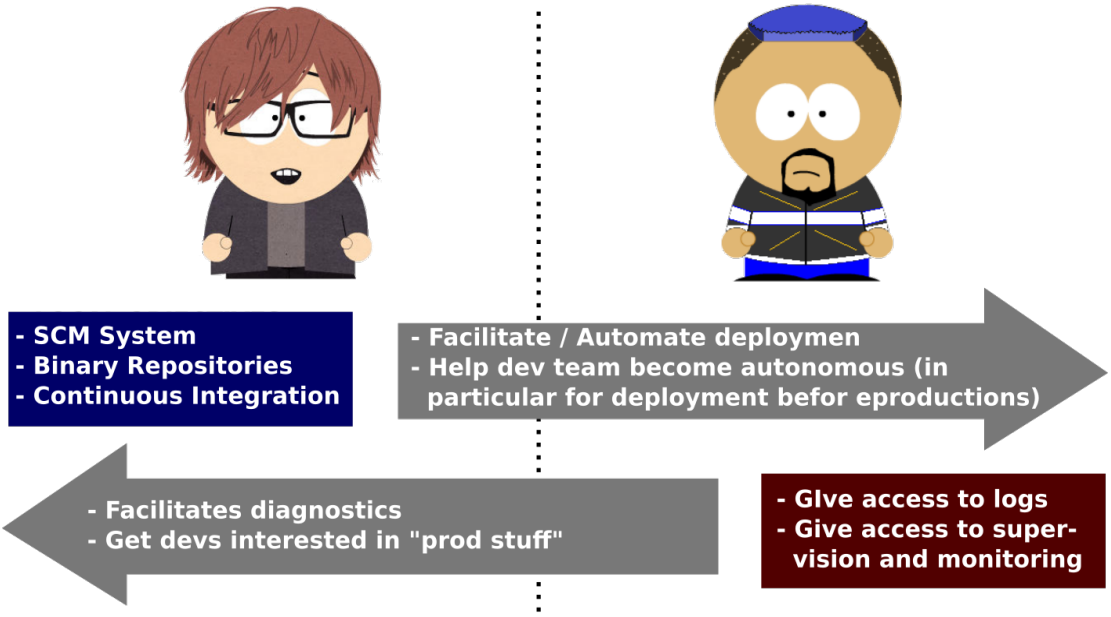
With DevOps, operators become fully integrated in the Software Development Process.

4.3 Share the Tools

In traditional corporations, teams of operators and teams of developers use specific, dedicated and well separated set of tools.
Operators usually don't want do know anything about the dev team SCM system as well as continuous integration environment. They perceive this as additional work and fear to be overwhelmed by developer requests if they put their hands on this

systems as well. After all, they have well enough to do by taking care of production systems.
Developers, on their side, usually have no access to production system logs and monitoring tools, sometimes due to lack of will on their side, sometimes for regulation or security concerns.

This needs to change! DevOps is here for that.



(Source : Mathieu Despriee - OCTO Technology - Introduction to DevOps)

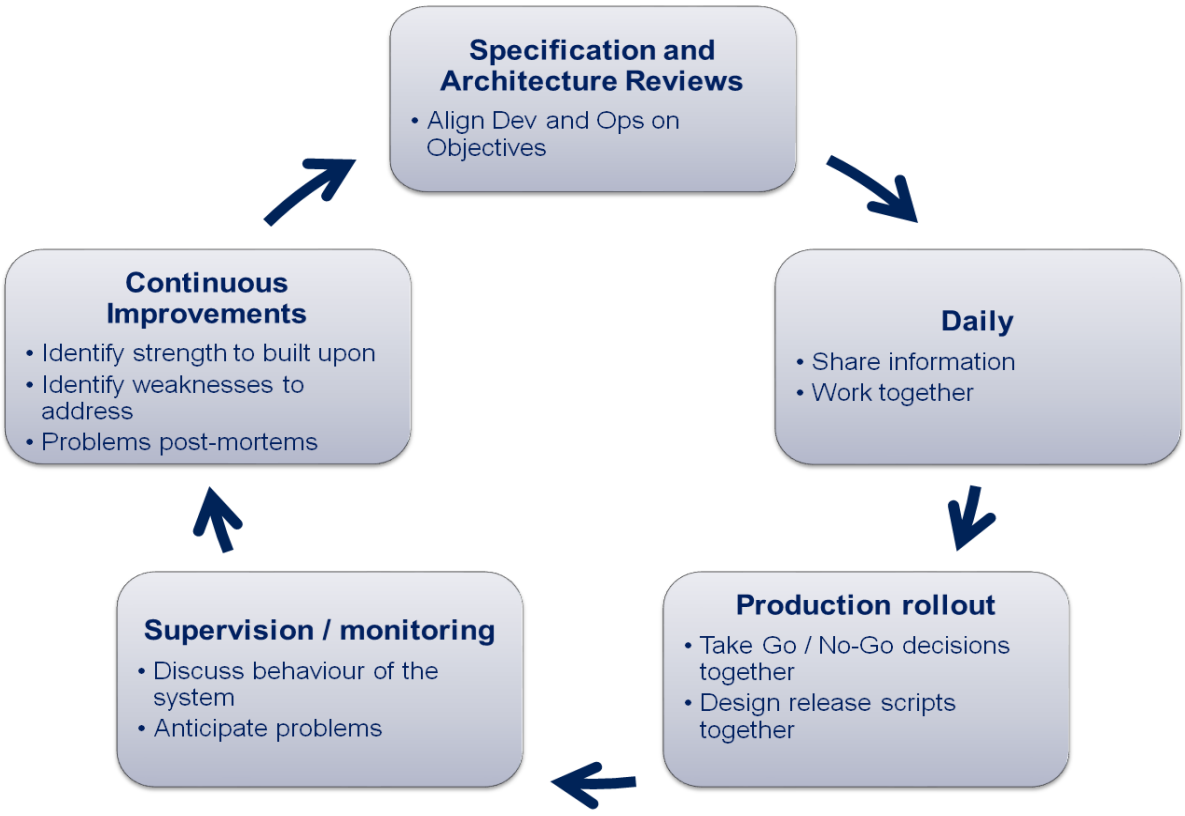
One should note that this can be difficult to achieve. For instance for regulation or security reasons, logs may need to be anonymized on the fly, supervision tools need to be secured to avoid an untrained and forbidden developer to actually change something in production, etc. This may take time and cost resources. But the gain in efficiency is way greater that the required investment, and the ROI of this approach for the whole company is striking.

4.4 Work Together

A fundamental philosophy of DevOps is that developers and operations staff must work closely together on a regular basis.
An implication is that they must see one other as important stakeholders and actively seek to work together.

Inspired from the XP practice "*onsite customer*", which motivates agile developers to work closely with the business, disciplined agilists take this one step further with the practice of active stakeholder participation, which says that developers should work closely with all of their stakeholders, **including operations and support staff**.

This is a two-way street: operations and support staff must also be willing to work closely with developers.



In addition, other collaboration leads:

- Have operators taking part in Agile rituals (Daily scrum, sprint planning, sprint retro, etc.)
- Have devs taking part in production rollouts
- Share between Dev and Ops objectives of continuous improvement

5. Conclusion

DevOps is a revolution that aims at addressing the *wall of confusion* between development teams and operation teams in big corporations having large IT departments where these roles are traditionally well separated and isolated.

Again, I've spent two thirds of my fifteen years career working for such big institutions, mostly financial institutions, and I have been able to witness this wall of confusion on a daily basis. Some sample things I got to hear:

- "*It worked fine on my Tomcat. Sorry but I know nothing about your Websphere thing. I really can't help you.*" (a dev)

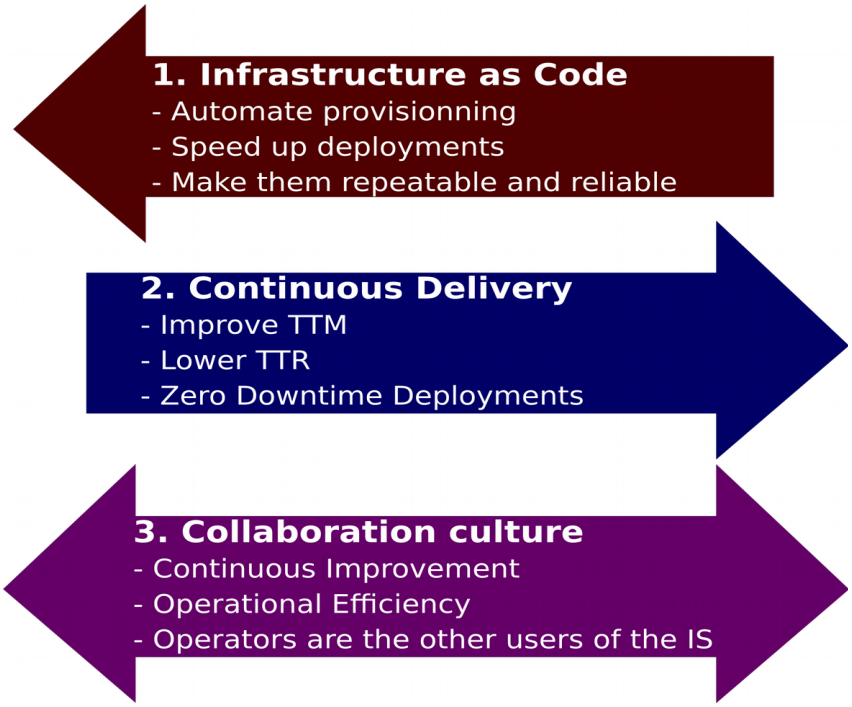
- "*No we cannot provide you with an extract of this table from the production database. It contains confidential customer-related data.*" (an ops)

And many more examples such as those every day .... every day!

Happily DevOps is several years old and increasingly even these very traditional corporations are moving in the right direction by adopting DevOps principles and practices. But a lot remains to be done.

Now what about smaller corporations that don't necessarily have split functions between developers and operators?
Adopting DevOps principles and practices, such as deployment automation, continuous delivery and feature flipping still brings a lot.

I would summarize DevOps principles this way:



DevOps is simply a step further towards Scaling Agility!

Part of this article is available as a slideshare presentation here :
https://www.slideshare.net/JrmeKehrli/devops-explained-72664261