DEMYSTIFYING

The Guide to Overcoming DevOps Challenges

DevOps

Sam looked out the window and sighed. Another late night at the office after **last week's disaster**. As project manager on her company's software development team, it was her job to make sure everything went according to plan. This wasn't the first time **a software project failed to deliver**. She wanted to do better. She didn't want to see her team repeating the same mistakes.

Sam stayed late that night to work on a post-mortem evaluation of the project disaster.

How did all of these problems happen all at once? Apparently the operations team installed the wrong version of some third party component. After the software update, **everything collapsed**. Mike, the operations lead, was able to quickly install the correct version of the component, but then the client started seeing bugs in the software. The testing team had overlooked a critical part of the system because it wasn't included in the specifications. The developer claimed he had to rewrite that part so he could reuse some existing code. Unfortunately, it wasn't easy to fix, and rolling back to the previous version took hours because the database updates had to be reverted manually. **Ultimately, the client couldn't work for an entire day due to the outage**.

Sam thought for a moment. **Where did it go so wrong?** Why did the tester fail to test such an important component? How did the administrator install an incorrect version?

Sam knew that **poor communication and process was at the center** of all this. These teams could do great work, but they didn't always work together seamlessly. She'd be surprised if the people involved even knew each other's names. Add to that all the manual work that was required for a release and there were plenty of things that could go wrong.

She noticed a book half-hidden under some papers on her desk. ***DevOps Handbook***, the title said. She recalled picking it up at a conference a few weeks back. A friend who had already read the book and applied some of the DevOps practices said it addressed similar communication and process problems at his company.

She picked up the book and went through the table of contents, looking at the chapter titles: "**Culture**," "**Continuous Integration**," "**Automation**"... Sam skipped the page and thought, "Perhaps they were ready for — no, needed — **a paradigm shift**."

What is DevOps?

Sam was a bit nervous. She had to explain to company management why the team had failed to deliver last month and how they would prevent these problems in the future.

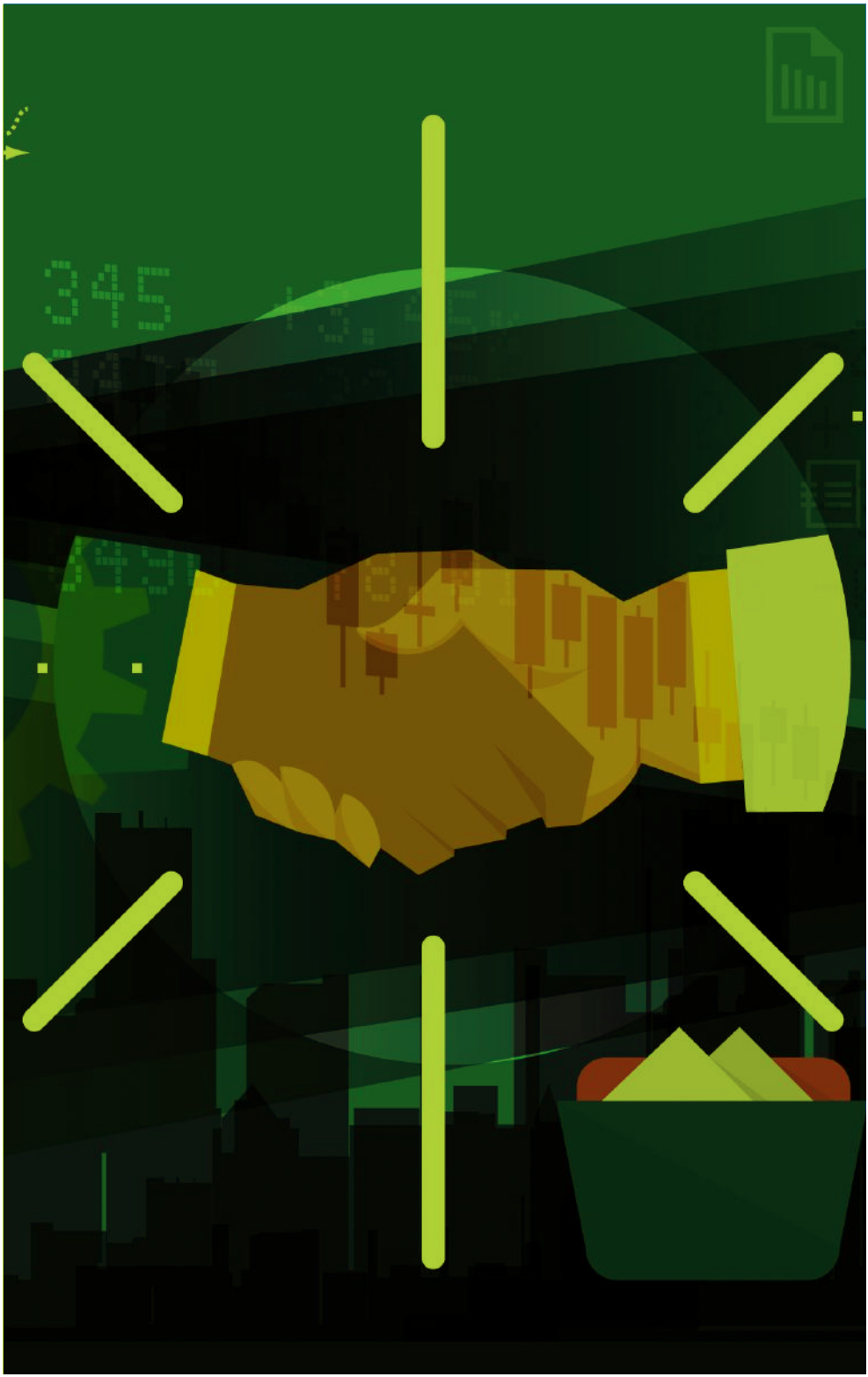
The team updated the software last week, after three weeks of overtime. Luckily, they deployed the second update without much problem. However, the update was weeks late and, because of the failed first attempt, the customer suffered some unforeseen financial losses. Apparently, it cost her company a bit of money to keep the customer from leaving.

After reading the book, Sam knew that the company had to change some of its practices — perhaps even their culture — if they wanted to avoid similar fiascos in the future. Sam suggested that the team embrace "**DevOps practices**" as part of the solution. And every time she mentioned it, the response was: "DevOps? What's that?"

DevOps, she replied, is "**the process of removing friction between the developer and customer**."

Although an accurate and succinct description of what DevOps can do, her team needed a more prescriptive explanation. They needed to understand what they could do to avoid friction between botched releases and customers in the future.

So Sam thought back to what she'd read and started with a **little history...**



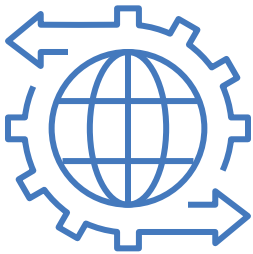
The History of DevOps

The term **DevOps** was first coined over a decade ago. Software developers and other IT professionals who had to write and support code were unsatisfied with the traditional method of software development. Developers and other IT teams were usually separate teams, sometimes in separate buildings, often with separate business goals and metrics of success. Getting software written and deployed could be frustrating for all teams involved, even more so when there were conflicting interests.

With new technologies, such as virtualization, cloud deployments, and containerization, both IT professionals and customers have different expectations. Everything has to work seamlessly, without downtime, and on-demand. For developers, this means they need instant access to the various resources needed to run their code. In ye olde world, this was simply not possible, as testers and system administrators were on different teams who safe-guarded their domain. The result was unhappy customers.

What began online in 2009 is now a trend in the industry with DevOps conferences around the world. With DevOps and the right supporting tools, teams can deliver value fast. A change request from the business can be in production in a matter of minutes.

When fully realized, DevOps enables non-disruptive innovation at scale, anywhere in the world.



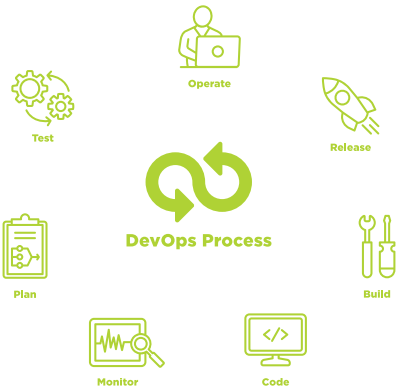
The Basics of DevOps

DevOps is a set of practices focused on collaboration between developers, testers, and system administrators or operations. In the broadest sense, it's the process of improving coordination and collaboration to produce better, more reliable products and to achieve business goals more quickly and reliably.

Traditionally, development and IT teams were formed as silos. Developers wrote software and handed it over to the testers, who then passed it on to operations. And sometimes, if you were really lucky, the security team may have given it a passing glance before it was rolled out to production.

By bringing developers, testers, security, and operations together in **cross-functional teams**, **better collaboration is possible**. DevOps brings organizations away from outdated corporate silos, increases flexibility, efficiency, and productivity. DevOps practices help operations and development work together through the entire software lifecycle, from design and planning to development to production support.

By practicing DevOps, companies **improve collaboration between teams**. They will be able to do more with less by automating wherever possible. Using a cloud-smart mindset helps to distribute workloads to the optimal infrastructure to meet business needs. **Adopting Agile methodologies** decreases the time to market while increasing stability and quality.



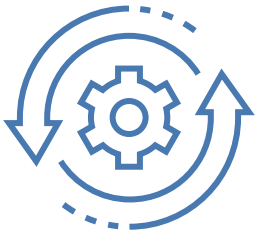
DevOps tooling

Modern developers expect the kind of tooling that makes collaboration for **Continuous Integration and Continuous Delivery (CI/CD)** easy. With Scrum boards, CI/CD pipelines, microservices and containers, Infrastructure as Code (IaC), and monitoring tools, it's getting easier to have near-real time software updates. With the right tooling and expertise, teams can ship multiple updates a day without the customer noticing, instead of a single update every few months that requires planned downtime.

With tooling that provides issue and work tracking, revision control and build automation, teams have **everything they need** to collaborate on software projects, from building pipelines for automated builds, tests, and deployments to tracking bugs and features for the next round of development.

Because this tooling makes **automation easier**, it becomes possible to automate builds, tests, and deployments. Public and private clouds play a big role in this. Cloud environments support Infrastructure as Code (IaC), which makes it easy to deploy resources, scalability, and high availability out of the box. Where these things are difficult or very expensive in traditional on-premise environments, they're the standard in modern public and even private cloud environments.

DevOps is ingrained in the culture of the company practicing DevOps. Usually, this isn't a direct, top-down approach. It's more likely that new practices are added to teams, practiced, and, if successful, then adopted as culture. Incorporating **these practices is the way forward** to competing in a modern, high-velocity business environment.



Forming cross-functional teams

Sam stood in front of an audience again. This time it wasn't management that was in front of her, but the full IT staff.

"Our management team has asked me to lead an effort to incorporate DevOps practices," Sam started. **"We're going to change some of the ways we do things around here."** Her presentation to management went well and only two weeks later she was talking to her teams about the coming changes.

"We're going to start by **forming new teams**. Each team will have some developers and at least one tester and one system administrator. Security will also be part of the review process from the beginning, not just an afterthought. We're also adopting a messaging platform that lets everyone be in touch with each other regardless of team or office." Sam expected some resistance, but she was prepared. To her surprise, there was some noise, but no one was really opposed to the idea.

"Does that mean," a developer started, "that whenever I need something installed, I can just ask my new team member?"

"Yes," said Sam, "**the idea is that we'll all work together on the same goals**. In fact, the admin in your team should already have that software installed by the time you need it."

Now, some of the administrators started to speak up. "But we're already busy enough as it is!" one objected. "We can't do our job and help the developers anytime they need something," another said.

"The **top priority** is to help your team of developers and testers, then work on the rest of your regular duties," Sam answered. "If all goes well, you'll spend **less time arguing with developers and there'll be fewer mistakes** — unlike last month — leaving you with more time to fix other issues. The same goes for the developers and testers. I expect you to find and fix issues sooner because you're working together in the same team towards the same goal. Here, I have a list with the new teams. Starting Monday, your new office is across the hall. After we settle down, we're going to define some sprints and **work towards more iterative goals.**"

Culture

Probably the **hardest part about DevOps is changing the company culture**. Having all the tools and frameworks isn't going to help if the people are not committed.

One step towards successful collaboration is sharing. Developers, however, can be very protective of their code. Likewise, admins don't like to give others access to resources, sometimes not even log files that developers need to track down bugs. Often the testing and the security audits just get missed because there isn't time and they slow things down. If a problem occurs it's "not my fault" and the finger pointing begins.

Once **people take shared ownership** for software and the environment it runs on, its code quality and security, a team can look for solutions together. If a team deploys a bug into production, it's not the tester's fault for not finding the bug, it's the team's fault because they share the responsibility for the application.

In a DevOps team, it's not important who gets the blame. It's important that the team fixes the bug as soon as possible, even if the original author has a day off.

This is, of course, most effective if the cultural change is not limited to the technical teams. Technical teams often attribute failure to the business for asking impossible features or to management for getting insufficient budget or time. Likewise, the business blames the technical team for buggy or counterintuitive software or software that's missing features. Whoever's to blame, the fact remains that the software isn't working as it should and it should be fixed. And **fixing software is a team effort**.

Bugs and faults happen, but they must be identified, prioritized, investigated, and plans made for remediation. It's important that teams treat this not only as work, but also a process improvement to cover gaps.

The mindset progresses from the traditional name-and-shame in the heat of the moment to adopting blameless post-mortem process, to **establishing retrospectives that lead to prioritized work scheduling**.

Being Agile

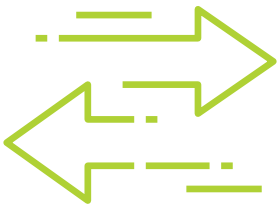
Back in the day, project teams used the Waterfall method to develop software. Analysts worked out all the specifications. Developers would build for months, or even years. Then everyone held their breath while the product was deployed.

This method has a few pitfalls.

First, specifications can change and frequently do. Things that were true last year may be changed or obsolete now. Since software was only deployed when it was finished, there wasn't a lot of room for change. Also, because entire applications were deployed at once, teams had to plan releases ahead of time because they could cost hours or even days to complete.

A lot of that has changed. With newer **Agile methodologies**, **teams plan for change** — sometimes on a daily basis. Instead of building everything in one go, **teams make incremental changes to software that is released regularly**.

One Agile method is **Scrum**, **where teams work in "sprints"** — cycles of two or three weeks. A product owner creates and prioritizes tasks, also called user stories, in a backlog. In each sprint, the team commits to some user stories that they will complete in the coming sprint. These will typically be the user stories with the highest priority. When stories are unclear, the product owner can ask the business for clarification so that all stories are clear before they are committed into a sprint. After each sprint, and sometimes during the sprint, the team deploys the work into production.



By working this way, **teams can quickly react to changes** in the business. For example, when the rules for invoices change, the product owner can create a user story and put it on top of the backlog, giving it the highest priority. The team can then commit to it in the next sprint. In some cases, the team can even change the current sprint by dropping the lowest priority story in favor of a new story with higher priority.

Because **teams are cross-disciplinary**, everyone is involved in estimating and planning the user stories. When a story requires third-party software to be installed and existing software to be modified, an admin and a developer can work on the same story simultaneously. Likewise, when a developer mentions that a seemingly unrelated part of the software has to be rewritten, the tester can take this into account during testing. Once testing is complete security can run an automated audit to make sure there are no holes where data can be hacked. **This approach improves collaboration, time to deliver and, ultimately, the quality of the software**.

Automate whenever possible

“Alright team,” Sam said, “you’re all doing great work and I think we can up our game.” The developers, testers, security, and administrators were now forming **agile**, **cross-functional teams**. Collaboration had already improved in the past few months. “Bugs are being found and fixed and operations have a much better idea of what’s necessary for the code to run. Let’s discuss the next step in improving our quality, continuous integration and deployment.”

“We’ve already taken a look at Jenkins for automated builds and tests,” one of the developers said, “it looks great.”

“This build system allows us to work with multiple clouds,” another developer added, “should we ever move to the cloud.”

“Good,” said Sam. “Management is talking about moving to hyper-converged infrastructure (HCI) to give us more cloud experience internally, but don’t worry about that for now. Let’s first focus on **improving the quality of our software** so that if we move to the cloud we’re ready.”

“One question,” said one of the system administrators. “It’s fun and all, automating everything, but what are we going to do with automating our infrastructure and our databases?”

“Perhaps it’s best if we leave that be for now,” Sam said. “We can automate our virtual machines,” another administrator said, “but I don’t trust automated tools with our data.” “And you trust people who make mistakes?” another asked.

“Well, people can think and decide whether an update will work or lock up the entire database.”

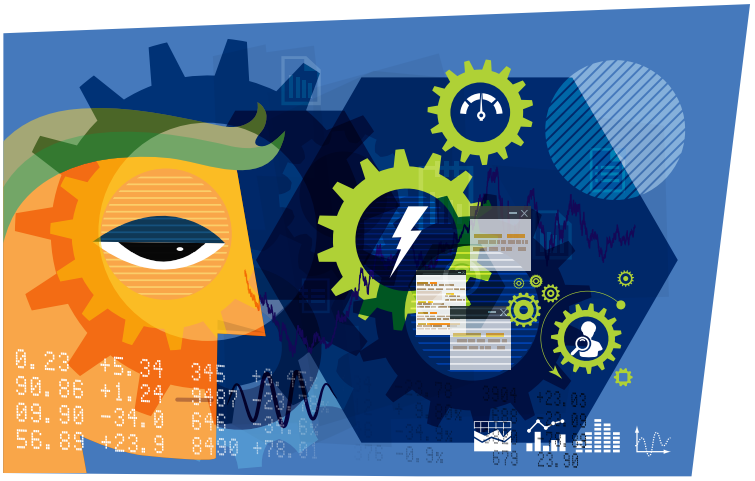
“Yeah, but they can also make the wrong call, or accidentally delete a table, or worse, a database. It wouldn’t be the first time.” A couple of people laughed while looking at one of the administrators.

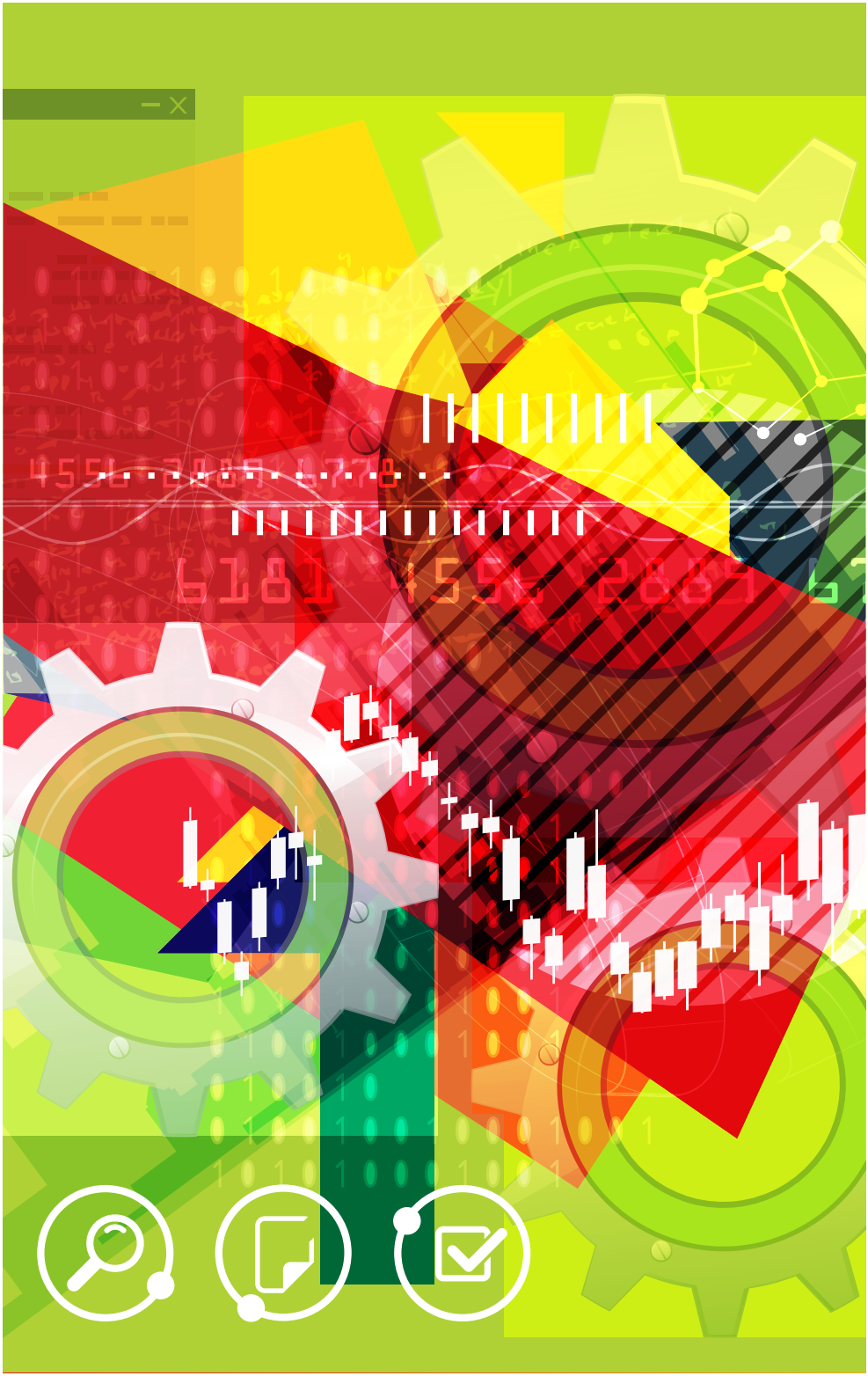
“Yes, yes, people make mistakes. But so do computers, I guess.”

“Alright,” Sam interrupted. “We can have the best of both worlds. Automating and checking in our scripts via Git will allow us to back out changes and see what went wrong if something goes wrong. We should treat our automation scripts the same way we treat our other code. Let’s see what we can do for **our infrastructure**: we should be able to **completely automate the deployments of new services**, even if they require new machines to run on.”

“Agreed, we’ll do some research.”

“Team, this is really coming together.”





Continuous Integration

Continuous Integration (CI) is the process of merging all developers' changes into a **single baseline**. Developers commit their changes into source control, preferably on a feature branch, which then triggers tooling that builds and tests the new code. CI tooling can automatically merge code back into the main branch if the code is correct. It can even incorporate optional, manual code reviews into the process. When a build or test fails, the software is unhealthy and developers should fix it as soon as possible.

By using CI tools and techniques, teams get **near-instant feedback** on their code. Gone are the days when a developer quickly checks in code before heading out for the day, only for the next developer to find out the next day that the code no longer builds. With a version control system such as Git, teams can **easily build code** on separate branches so that even if their code fails to build it won't affect other developers working on other branches. Perhaps even more important, it won't affect the main branch, which means the software could be **released at any time**. Only when a feature branch is built, tested, and reviewed should it be merged back to the main branch.

It's a best-practice to use pull requests and code reviews. That means that code can never be committed directly on the main branch. Team members **share knowledge and feedback** if they have to manually accept changes into the main branch. Also, team members can find technical bugs before they reach the main branch or even a tester. By writing tests and automatically testing code, a certain level of **correctness can be guaranteed**. The quality of the software relies on the quality and coverage of the tests. There are varying levels of tests, such as unit tests, which test units of code; UI tests, which test whether a user interface works as it should; and integration tests, which can test whether the connection between various components work as they should.

After testing, running an automated security audit makes sure that the code doesn't have a gaping whole where data could be stolen. Automated testing can be a good addition to manual testing, but manual testing should not be abandoned altogether. A system can work fine, but still not be what a customer asked for. Only manual acceptance testing can verify if a system does what it should.

Continuous Delivery and Deployment

One step further than automatically building and testing code is to **deploy it automatically**. There are two levels of automation here, **Continuous Delivery** and **Continuous Deployment**. (These practices share a common acronym, so make sure you know which practice folks are discussing when using "CD.")

Using **Continuous Delivery** requires someone to manually press a button before software is released. This manual action is eliminated with continuous deployment. In theory, a programmer could commit some code and the CI tooling could build it, test it, and **release it into production without human intervention**.

More common is that some environments, like development and test, use **Continuous Deployment**, while acceptance and production use Continuous Delivery. This approach allows businesses to **test new functionality in a controlled environment**.

Deploying software is one thing, deploying the infrastructure the code has to run on is another. It is impossible to automate the process of deploying actual servers because someone has to order it, unpack it, install it, and so on.

By making use of virtual machines (VMs), automated deployment becomes an option. VMs can be deployed and configured by using tools such as Chef, Calm, or Puppet. However, one of the hardest things for IT teams is deploying on-premises infrastructure and as a result this task is often not automated.

For enterprise grade virtualization options, **Nutanix offers Acropolis and AHV**.

Deploying databases

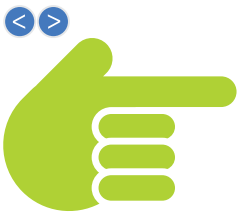
One of the scariest parts of CD is automatically deploying databases. There are a couple of options here.

First, there are **code-first ORMs** (object-relational mappers) that allow developers to **code their database objects** as plain classes in the language of their choosing. The ORM can automatically migrate the database to the newest version when the application starts. This solution has limitations, and because neither developers nor operations have a lot of influence on this process, it's not very feasible for big projects with big data.

A second option is to still use code-first, but instead of letting the software handle it, **generate scripts using ORM CLI tools** (provided one exists for your preferred ORM) in your build pipeline, and run those scripts against the database in your deployment pipeline. The **added benefit** is that the application doesn't alter the database and you can run additional scripts more easily. Both methods rely on the script generation of an ORM, though.

A more viable option is to write your own scripts and run those using a database migration tool in your deployment pipelines. One such tool is **Nutanix Era** which also offers other capabilities like **one-click provisioning and copy data management**.

A solution to this migration problem is using **schema-less NoSQL databases**, such as MongoDB, which do not require schema updates. With schemaless databases, developers can simply insert records and expect certain fields when querying. One of the problems with this approach is that it's more error-prone and requires developers to not only learn a new tool, but to think about data storage differently.



10

The cloud-smart mindset

This time it wasn't Sam who did the talking. Her team invited her to this meeting because they wanted to make use of the cloud as soon as possible.

"We've made good progress on automating builds, test and deployments," one of the administrators started, "but we can't script extra hardware."

"We can spin up new virtual machines and even install and configure some software, like IIS, but it's pretty difficult and time consuming." "And with that new order we've got coming up, scaling is going to be an issue."

"Yeah, at this rate it looks like we've got to build an entire new data center exclusively for handling peak demand. Those servers will be sitting idle most of the month."

"I see," said Sam, "the cloud came up in management meetings, but they're not quite there yet. In particular, **they worry about security and reliability**."

"How so?"

"You're never really sure where your data is and if it's properly secured." "You can pick the region where you deploy your services. Furthermore, these cloud vendors thrive on trust, and trust can only be gained by being completely secure. They have these internal teams trying to hack into these services to guarantee there are no back doors. **We can't compete with that level of security.**"

"Yes," said Sam, "and without an internet connection most of us wouldn't be able to do our work anyway. And we do have occasional issues with our own servers as well."

"**The cloud just offers so much in terms of flexibility**. We can script our entire infrastructure, enable auto-scaling and high availability, and instead of having a data center that's sitting idle most of the time, we can just spin up some extra compute power when we need it." "Alright," said Sam. "I'll talk to management."

Hybrid cloud and multi-cloud

The cloud has a couple of benefits. It offers Infrastructure as a Service **(IaaS)** and Platform as a Service **(PaaS)** solutions. That means you don't need your own infrastructure or you'll need (significantly) less of it. When going for PaaS — or cloud-native — solutions, it also means you'll have a lot less maintenance because the cloud provider handles that for you. **The cloud has built-in support for scaling on a level that could never be achieved in an on-premises environment**.

Let's say you have a web service running on an on-premises web server and you want to migrate it to the cloud. A so-called "lift-and-shift" solution would be to deploy a virtual machine (IaaS) in the cloud and host the web service there. The downside here is that you still need to manage your VM, like updating the OS and any installed software.

A different solution would be to deploy an app as a service (PaaS), which allows you to run a web service out of the box. You have no access to the OS (other than picking Linux or Windows) or the software installed on it. You can only configure a limited number of properties, such as SSL and certificates, but it's a lot easier than managing a VM. And since it's a managed service, it's easy to roll out using CI/CD.

Unfortunately, not every cloud can run every workload. Especially when you already have an on-premise infrastructure, it may be easier or cheaper to continue running certain workloads on-premise. **That's why multi-cloud and hybrid cloud strategies are becoming popular among enterprises**.

A hybrid cloud is a mix of public cloud (like Azure, AWS, or Google Cloud) and on-premise infrastructure, often with some third-party tool to orchestrate between the two.

Nutanix Clusters can help in managing hybrid cloud environments. Using a hybrid cloud you can **reap the benefits** of being in the cloud while still **running and connecting with your own infrastructure,** allowing you to **manage applications in private and public clouds** without retooling or rework translation between environments.

A multi-cloud environment is simply using multiple public clouds for different (and sometimes even the same) **services**. For example, a team could host one service in Azure and another in AWS. Additionally, a team can host a web service in both AWS and Azure with a load balancer between the two. That way, when one cloud provider has issues, the other will still continue to work.

By using multiple cloud environments, teams can host services in the clouds that are best suited for the current task. An added bonus is that companies don't have to rely solely on a single cloud provider, so they're not locked-in to a single cloud. Nutanix **Xi Beam** helps you keep your multi-cloud environment secure and cost effective.

Infrastructure as Code

One of the **biggest benefits of the cloud is that it's all scriptable**. Operations can deploy your IaaS and PaaS solutions using scripts. DevOps tooling can then easily deploy applications to the already existing services.

Traditionally, operations often had "pet" servers. Servers would get recognizable names like Zeus, Hades, and Ares (from the Greek mythology) or Bali, Lombok, and Sumatra (Indonesian islands). Each of these servers were unique and had their own "personality." For example, Zeus was the database server and Hades was the CI server. System administrators would manually install these servers for specific purposes. When one went down, everybody knew.

With the rise of virtualization, the "pet" server model gave rise to the "herd" or "fleet" server model. In the fleet model, each server acts pretty much as the next one and so they're given numbers, like trucks. Operations could deploy server001, server002, and server003 using automation tools like Chef and Puppet. When a server went down operations could just replace it with another.

In cloud computing, operations can automatically deploy VMs using CI/CD. By tracking metrics, such as CPU and memory usage, the cloud can automatically increase and decrease VMs based on performance. This is also called the "germs" server model. It makes **scaling your applications easy and automated**.

Next to VMs, it's possible to deploy other IaaS and PaaS services using scripts or code. Azure, for example, uses declarative JSON templates, named Azure Resource Management (or ARM) templates, that can deploy almost any of the hundreds of Azure resources available. Likewise, AWS uses CloudFormation. There are also cloud agnostic tools like Terraform.

Using scripts to deploy infrastructure is also called Infrastructure as Code (IaC). An alternative for deploying and configuring services is using PowerShell or the specific cloud's native CLI, such as Azure CLI. The benefits of IaC and scripts is that they can be **added to source control and run automatically from anywhere**.

The benefits of Microservices

Because deploying software is becoming a lot easier, teams can deploy software a lot more often. This gives rise to **powerful new software architecture models, such as microservices**.

With microservices, an application is broken up into many smaller parts. For example, an application could have a sales module, an invoicing module, and a stock module. In a monolithic architecture, teams would create a single solution and deploy it all at once. With microservices, these could easily be six different applications.

The **benefit** is that teams can deploy parts of an application separately. Likewise, teams can replace parts of an application without touching any other parts. Teams can also host different parts in different clouds and they can use different databases. Because each part of the software is now running on its own, it can also be scaled separately.

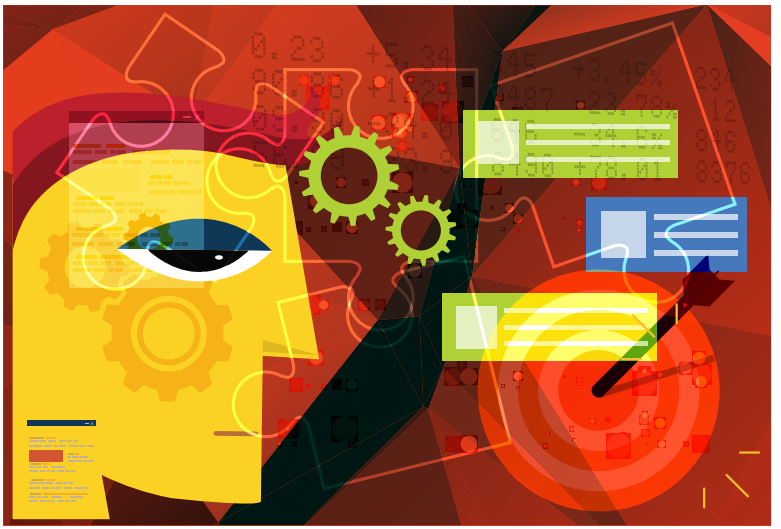
A challenge with microservices is managing so many different parts that must work together. Containerization solves this problem. Basically, a container is a lightweight VM that, unlike an actual VM, runs on your OS. A container runs a piece of software described in a template. The most popular container software by far is Docker, which makes use of dockerfiles.

The beauty is that when an application needs SQL Server to run properly, there is an image that will run SQL Server in a container, eliminating the need to install SQL Server locally. When the container stops, the app is gone too.

When using containers with microservices, you'll soon find yourself having numerous containers that all have to be started, sometimes in a particular order. **Container orchestrators can manage this**.

The most popular by far, supported by all cloud vendors, is Kubernetes. Using a declarative file you can indicate which containers should be started first, how many instances you need of a specific container, on how many clusters (VMs) it should run, and so on. When a cluster fails, Kubernetes will try to move the applications running on that cluster to other clusters.

Nutanix can help organizations set up and monitor their Kubernetes clusters with **Karbon**. With proper setup and monitoring, **your application scales automatically and becomes self-healing**.



Epilogue

It was time for the annual company holiday party. As always, the CEO officially started the party with a speech.

"Welcome everyone. I know we're all having a good time, but if I may have your attention for a little while, please. We're close to a new year and as usual I'd like to reflect on what we accomplished this last year and look forward towards the future.

"I'm sure no one has forgotten the catastrophic deployment we had last year. We took some damage and almost lost a very good customer. Looking back, it wasn't a matter of if it would happen, but when it would happen.

"While the people in this room are mostly the same people, this company is far from the same. What went wrong back then was an eye opener, a sign that we had to do things differently. And so we did. We started our journey into DevOps and we continued that journey this past year. **We are seeing great results**. I remember Sam's presentation about 'removing friction between developers and customer value.' She advocated a cultural shift, automation and cloud. So what has it brought us? "You'll see some numbers on the sheets behind me. We do deployments a lot more frequently, for some customers we even went from monthly deployments to multiple deployments per day. That means the time it takes from fixing a new bug **improved from a month, on average, to just days**. And despite this astronomical rise in deployments, we've had **fewer issues**. We've also started a couple of cloud projects and so far they're looking very promising.

"According to the annual employee satisfaction survey you've been noticing these changes too. You feel **more engaged with the company**, with customers, and with each other. You've also indicated that **collaboration with each other has improved** a lot this past year. I am proud to be a part of this and I am proud of you, the people who make it possible."

The speech went on for a while, but Sam wasn't really listening anymore. She was proud to have pulled it off. Proud of her teams who managed to do it. She was already thinking of new ways to improve the development cycle.

Want to learn about how DevOps can **help your organization achieve continuous innovation at scale? Check out DevOps and Enterprise Cloud or visit Nutanix.Dev**.

NUTANIX™
YOUR ENTERPRISE CLOUD

To cope with the fast moving IT world, companies need to increase their pace, shorten work cycles, and improve delivery efficiency. To achieve this, IT teams need to implement a DevOps strategy to give their business a much needed boost. To learn more about a high-performance infrastructure that will help make DevOps a reality, check out **nutanix.com**.