rapidscale

DevOps on the AWS Cloud

- Solution Provider
- Managed Services Provider
- SaaS Services Competency
- DevOps Services Competency
- Security Services Competency
- Generative AI Services Competency
- L1 MSSP Services Competency
- Migration Services Competency
- Healthcare Services Competency
- Small & Medium Business Services Competency

Introduction

In order to meet the demands of increasingly agile software development practices, IT teams need to be able to deploy applications in a rapid, repeatable, and reliable manner. This has led to a dramatic rise in the popularity of the philosophy and practices of DevOps.

Amazon Web Services (AWS) supports many DevOps practices and can supply IT teams with a powerful toolset to automate and orchestrate infrastructure buildout, configuration, testing, and deployment. A well-executed DevOps approach enables IT teams to concentrate on the continuous improvement of processes that directly impact business agility, rather than on manual deployment and infrastructure updates.

The Core Principles of DevOps on the Cloud

The goal of every DevOps team is to consistently, safely, and quickly deliver changes into production or deploy new software. To accomplish this, infrastructure must be controlled and versioned like the software it supports. This can be described by four main principles:

1. Design for constant change
2. Build repeatable components, not snowflakes
3. Anticipate failure
4. Human effort = security risk

The first, and probably the most important, principle is to design for constant change. In modern development, the ground is constantly shifting underneath us, and we need to be comfortable with ongoing changes. That means building an environment that is not just suitable for Day 1, but can be managed efficiently and changed without too much risk.

Part of designing for constant change is not just servicing an individual stakeholder's concern, but doing it in such a way that issues can be resolved more efficiently if they come up again. That means designing repeatable components, not doing ad hoc or one-off builds or deploying just to get the job done.

The third principle we follow is to anticipate failure. On the cloud, you must design a system that rarely fails, but you also need to design a system that fails well. We leverage some of the Auto Scaling features AWS provides to make sure we have redundancy at every level of our stack.

And finally, human effort is a security risk. We try to avoid direct human intervention in an environment as much as possible and look for opportunities to avoid human error.

Infrastructure as Code

In order to build a system that embodies the above principles, you need to treat your infrastructure like a piece of software; in other words, your networks, storage, etc. are manipulated with code, versioned in a code repository, and continually updated and improved. Builds are repeatable and reliable, requiring minimal human effort.

It is frequently the case that in growing DevOps teams, the same rigor that is applied to application code development is not applied to infrastructure. Infrastructure is still provisioned manually, either in the cloud console or CLI. No mature testing procedures are in place, nor could changes be easily rolled back to a previous state. This is why infrastructure as code has dominated conversations about cloud operations in the last five years; systems engineers realized that in order to bring the positive effects of agile software development downstream to IT, they needed to change the fundamental practices of building and maintaining infrastructure.

The true impact of infrastructure as code can most easily be seen in an example. When we are creating a new environment or a new network architecture in an environment, there are two clear approaches. We could build things manually (in the AWS console) or automate. The differences are clear.

If we build the environment manually, the process is slow and requires a highly-trained engineer. As time goes on, it is likely that each environment will be built differently, with many custom configurations that require custom documentation. Each environment has its own deployment process, and likely its own "specialists" – so deployment is dependent on the availability of one or two key engineers. On top of that, there is a high chance of "forgetting" something important (like a key security configuration).

The second option is to build the environment by applying the principles of infrastructure as code, and on Amazon Web Services, this usually involves the use of AWS CloudFormation. AWS CloudFormation is a data format (JSON) that allows you to write code and output a foundational AWS environment. This means that a systems engineer could write a set

of templates for AWS environments that includes the company's baseline configurations and universal attributes, and then use those templates to launch future environments rapidly. The value of this approach is tremendous. Entire environments can be quickly duplicated, modified and deployed in quick time. Templates can be checked into a source code management system, linted and validated using tools and IDEs. AWS CloudFormation can also kick off bootstrapping through UserData, creating a seamless bridge into the operating systems where machine images and configuration management can take over. In addition, any infrastructure change can have an audit trail and proper process built around it, automated to reduce human error and disparate configurations.

The value of this approach is tremendous. Entire environments can be quickly duplicated, modified and deployed in quick time.

Over time, a company develops a collection of AWS CloudFormation templates for various use cases. A central IT office can even organize these templates and make them accessible to engineers in a controlled, self-service fashion through a tool like AWS Service Catalog. Through Service Catalog, project teams can access centrallyapproved templates that build out critical components, significantly accelerating build-out while simultaneously putting up guardrails to control core architectural standards. The company can also simply keep the templates in a code repository like Git.

Configuration Management

Configuration management (CM) is the next key element of DevOps in practice on AWS. CM is a 15+ year old technology that has come to prominence in the age of cloud due to its ability to programmatically "dictate" the ideal configuration of an operating system, and then maintain that configuration over time.

The fundamental purpose of configuration management is to deploy environments prescriptively. By defining everything on the servers in a single location, there is a single source of truth about the state of the entire infrastructure. The value of configuration management is that when you want to change the OS, you can change code on the servers without making major changes to the servers themselves. CM is also used to install various security features on an instance, like Identity Detection System agents, log shipping, monitoring software, etc. as well as requiring MFA and binding the instance to central authentication. That means changes can be rolled out to many instances at once — and be rolled back. When you are done, each server is in a known, ideal state. When you build and maintain your environment manually, change is slow, cannot be rolled back, and has a higher risk of human error.

On AWS, there are many possible tools to control configurations. You can choose to use an AWS native tool like AWS OpsWorks, which uses Chef. You can deploy your own server with Puppet, Chef, or another CM tool of your choice. Obviously, the latter provides greater control, but requires that you provision and maintain your own Puppet/Chef server. You can also manage services in a more restricted (but more fully-managed) way through Amazon EC2 Systems Manager, which is essentially a collection of a la carte services to create system images, apply OS patches, track system configurations, etc.

If your team is just beginning to develop DevOps capabilities, an AWS-native CM tool can be a good option. This is especially the case if you want to get a CM system up and running quickly; a tool like Amazon EC2 Systems Manager will let you pick and choose the controls that you need without having to build out an entire framework, and includes a State Manager function. If your team has existing expertise on a non-Chef CM platform like Puppet or Ansible, even if that experience is in traditional non-cloud infrastructure, then running your own CM server on AWS is likely the most flexible and robust option.

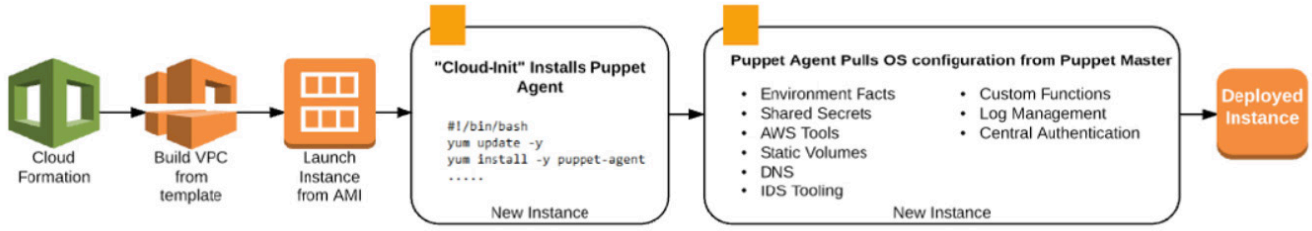
For the purposes of this e-book, we will discuss Puppet with the context that most configuration management platforms offer identical services with different term names.

Configuration Management and Auto Scaling

Auto Scaling, when correctly implemented, significantly decreases the risks associated with traffic overflow causing server failure. At the same time, costs are reduced. Instead of running instances based on projected (assumed) usage and leaving excess resources in place as a buffer, one can run only the resources that match actual usage, on a moment-to-moment basis.

Auto Scaling is a capacity of AWS, but not a built-in function. There are many ways to getting a system from zero to 100% scalability, and configuration management tools like Puppet are the most common tools. On any kind of infrastructure, automated software installs can save your team a great deal of time, but in a dynamic cloud environment, CM tools can actually create the AWS resources your nodes require to operate, such as Elastic IPs, network interfaces, or block storage. When using a tool like Puppet on nodes in AWS Auto Scaling groups, however, a few different issues can arise.

Puppet scripts can replace the need for a perfectly baked AMI. Instead, you can create a vanilla template with the minimum possible configurations that replaces your individual Golden Masters for each server role. In this scenario, your instance userdata or boot script needs only to do what is necessary to connect to the Puppetmaster.



RapidScale's instance build process. Every instance follows the same process, so there are no "snowflake" systems.

The final step is integration between deploy process and Auto Scaling — where Puppet scripts automatically integrate those Amazon EC2 instances into Auto Scaling groups. Puppet scripts can replace the need for a perfectly baked AMI. Instead, a vanilla template with the minimum possible configurations replaces individual Golden Masters for each server role. In this scenario, instance userdata or boot script needs only to do what is necessary to connect to the Puppetmaster.

Continuous Deployment

Deployment automation enables continuous integration, where development teams deploy code to production several or even hundreds of times a day. With every commit that is pushed out to the site, a whole suite of unit tests verify what is working. If a team has a one-button process to push from Git, so they can deploy a large update to just one instance and then run statistics on that one node to make sure it is working correctly, this is a very mature systems development process.

Hundreds of tools exist to partially or fully automate a CI/CD deployment pipeline. Amazon Web Services' automated code delivery tool, AWS CodeDeploy, is a script with an agent, a web interface, and a code repository. Its simplicity and flexibility are the reasons RapidScale recommends AWS CodeDeploy to enterprises on AWS.

For most enterprises, the best deployment tool is the one that does not require significant application refactoring. The great thing about AWS CodeDeploy is that it is extremely flexible; it is platform and language agnostic, works with any application, and the process that enterprises currently use to deploy software can be slotted into AWS CodeDeploy's framework. This flexibility is due to AWS CodeDeploy's AppSpec configuration file, where you can specify commands to run at each phase of deployment such as code retrieval, code testing, etc. These commands can be written in any language, meaning that if you have an existing CI/CD pipeline, existing stages can be modified and sequenced in an AppSpec file will minimal effort. You can also integrate CodeDeploy into your existing software delivery toolchain using the CodeDeploy APIs.

AWS CodeDeploy gives you the ability to do a rolling update across a group of instances, so that an update can be deployed to a specific number of instances for testing purposes and not to others. Deployment Health Tracking monitors the success of each instance update so that in the case of failure, you can pinpoint the exact instance and script experiencing failure. For systems engineers, this transparency into what developers are doing is crucial. When problems crop up, they have the ability to see what happened and why things went wrong. Everyone wants a predictable, controllable, transparent deploy process. AWS CodeDeploy gives you greater control over which version of your code is running and where than a pure Configuration Management-run deploy does.

Deployment on AWS can happen one of two ways: either an engineer pushes a code revision, or an Auto Scaling event happens and a new instance is created, which needs to receive the latest version of your code. AWS CodeDeploy integrates natively with Auto Scaling. This means that if your instances are part of a Deployment Group and a new instance is created, Auto Scaling will wait for AWS CodeDeploy to "interject" with the application code before it puts the instance behind an Elastic Load Balancer.

Integrating Auto Scaling with a custom toolchain can be challenging without AWS CodeDeploy; you need to pause the scaling event, make sure the instance does not get added to the load balancing pool before it is ready, and then trigger this event after testing. Engineers will often use a configuration management tool like Puppet to orchestrate these resources. Puppet and AWS CodeDeploy work well together to spin up and bootstrap and instance, then kick off AWS CodeDeploy to add application code. Puppet can even do the work of installing the AWS CodeDeploy agent on each instance.

Microservices and Containers

The vast majority of enterprises have hundreds of applications, many of which are dependent on a common set of app infrastructure, packages, or "bridge" applications. When the applications work smoothly, this isn't an issue. In the event something fails, though, discovering the origin of the can be difficult when application boundaries are not discrete and enterprise teams have no map of application dependencies, leading to significant time spent combing through logs to discover the basic architecture of the program. Any change, even a small one, requires rebuilding and retesting of the entire application, because a small change may have cascading effects.

In the context of these difficulties, it is easy to see why the microservices approach is growing in popularity – even in large enterprises. Microservices describes an approach where a single application is built as a set of smaller services, where each service runs its own processes and can be deployed independently or as a group of services.

Docker is a software solution to effectively deploy microservices. Containers describe and deploy the template of a system in seconds, with all infrastructure as code, libraries, configs, and internal dependences in a single package, so that the Docker file can be deployed on virtually any system. By building in dependencies, Docker containers reduce inter-cloud operability concerns; apps that run well in test environments built on AWS will run exactly the same in production environments in on-premises data centers.

Containers and virtual machines both emulate multiple hardware systems that are isolated from each other, but VMs each have a full operating system while containers share the host OS. The time and resources saved by leveraging containers can have a big impact on application density. Some estimates show you can run twice the workload using containers than Xen or KVM VMs. Containers allow for much greater server density by removing redundant OS elements from the containers themselves. We have not tested the limits of application density. It is not complicated: you are sharing a Linux kernel (maybe XMB) so instead of having X times the number of VMs on your hardware, you are running 1 times X. These estimates depend on how big the kernel is plus the size of your base image.

Once Docker is in place, it drastically simplifies and de-risks the deploy process. Developers have more of a chance to work on applications knowing that once they deploy to a Docker file, it will run on their server.

Once Docker is in place, it drastically simplifies and de-risks the deploy process. Developers have more of a chance to work on applications knowing that once they deploy to a Docker file, it will run on their server. They can build their app on their laptop, deploy as a Docker file, and type in a command to deploy it to production. To take a very simple example: upgrading your OS in a monolith can be a multi-day or week process. Why? Because you have to understand which part of your application depends on which now-outmoded models. It's much easier to update to the latest version of your OS when you can update it in a single module (or in a single Docker base image) that you know will not have ramifications beyond the boundaries of your module/ container and then deploy that base image to a single Docker cluster (perhaps 10% of your compute power) to maintain availability while testing the new OS.

AWS's container orchestration service is Amazon EC2 Container Service (ECS). Amazon ECS takes away some of the configuration you need to complete with Docker, and allows you to easily run applications on a managed cluster of Amazon EC2 instances. Although Amazon ECS eliminates the need for you to install, operate, and scale your own cluster management infrastructure, you still need to set up and manage the underlying Amazon EC2 instances in the same way you would manage any Amazon EC2 instance – so Docker and Amazon ECS work best in environments that have already developed sophisticated configuration automation practices and where engineers have invested time in developing templates to describe cloud resources (AWS CloudFormation). You can achieve workflows where Jenkins or other configuration integration tools run tests and AWS CloudFormation scales up an environment, all in minutes.

There are many alternate container management services that you can use to orchestrate Docker on AWS. Kubernetes is perhaps the most popular alternative, likely because it is the best supported orchestration layer for traditional infrastructure and can work across any cloud or non-cloud platform. Docker Swarm and Mesos are also popular options. In our experience, Amazon ECS is the best option if you are running on AWS only.

While Docker is an extremely powerful technology, it does not automatically create microservices processes or teams. Before you start using Docker, you first need to decide if you want microservices with all of its benefits and risks. Then you need to travel the difficult road to build or refactor microservices applications, of which Docker is only a small part. Building your application to conform to a microservices model or merely with well-defined module boundaries is by far the more important and process-transforming effort.

Secure DevOps and Security by Design

In the early phases of DevOps adoption, developers and security teams sometimes work at odds; DevOps teams want to move fast, and security teams need weeks to test and integrate standard security tooling. This has led to the rise of a new approach to security in a DevOps environment, which is sometimes called DevSecOps, Secure DevOps, or more simply "DevOps done right," where security teams are involved earlier and more frequently in the infrastructure design and software development process.

In practice, this means that engineers need to formalize infrastructure design and automate security controls so that security is built into every part of the IT management process. This approach is often called Security by Design (SbD), meaning that security controls are imbedded in the IT management process rather than relying on the typical protective or reactive 3rd party security tools. In practical terms, this means that your engineers spend time developing software that controls the security of your system in a consistent way 24×7,

rather than spending time manually building, configuring, and patching individual servers. SbD is about coding standardized, repeatable, automated architectures so that your security and audit standards remain consistent across multiple environments.

The technologies described in this e-book – architecture automation, configuration management, deployment automation – can be used to create a system where security controls are consistently adopted and updated like a piece of software.

Architecture automation in AWS CloudFormation and configuration management ensures that your standard network architecture and Security Group rules are maintained. The #1 reason to automate security is that human memory is limited. The bigger the infrastructure, the easier it is to forget to close XYZ security loophole when launching a new environment, or remember to require MFA, etc. Engineers are a smart group, but automation created by expert engineers is smarter. Even something like inconsistent or sloppy naming is a bigger security risk than most people think. It would be fairly easy to mistake one security group for another if they are all called "launch-wizard-1" or "launch-wizard-7".

Similarly, when different environments are built by different engineering teams at different times, manual security configurations often mean custom configurations. This makes it very difficult to gauge the impact of a feature change on security. A single or limited number of custom configurations not only reduces the risk of unexpected security implications, but also means your team is not relying on the memory of the one or two engineers that built the application's infrastructure.

Ideally, we need a system that empowers developers and engineers to work in an agile fashion while still maintaining security and compliance standards; we need a toolchain that:

1. Makes it easier to build out compliant environments
2. Provides guardrails to prevent engineers/developers from launching resources outside of compliance parameters
3. Provides ongoing documentation about the configuration of infrastructure resources

The toolchains we have already described — templating, configuration management, monitoring — allow us to launch new compliant environments trivially and ensures very limited access to the environment and full documentation on every change. Together, this means a greatly reduced risk of undocumented configuration change, error, or lack of adequate knowledge about where sensitive data lives, and therefore greatly reduces the risk of compliance violations.

The SbD approach also has significant positive impacts on compliance efforts. The hardest thing to achieve in infrastructure compliance is not getting security and logging tools set up and configured, it is maintaining those standards over time.

In the old world, systems changed infrequently with long lead-times, and GRC teams could always spend 2–3 weeks evaluating and documenting change manually (usually in a spreadsheet). On the cloud, when code gets pushed weekly and infrastructure is scalable, this manual compliance approach can severely limit the success of cloud projects, slow down DevOps teams, and frustrate both business and IT.

When systems are complex, there must be an equally powerful set of management tools and processes to enforce and maintain configurations. Continuous compliance is only possible if you treat your infrastructure as code. If your infrastructure can be controlled programmatically, your security and compliance parameters are just pieces of code, capable of being changed more flexibly, versioned in Git like any piece of software, and automated to self-correct errors. This is the future of any type of security on the cloud.

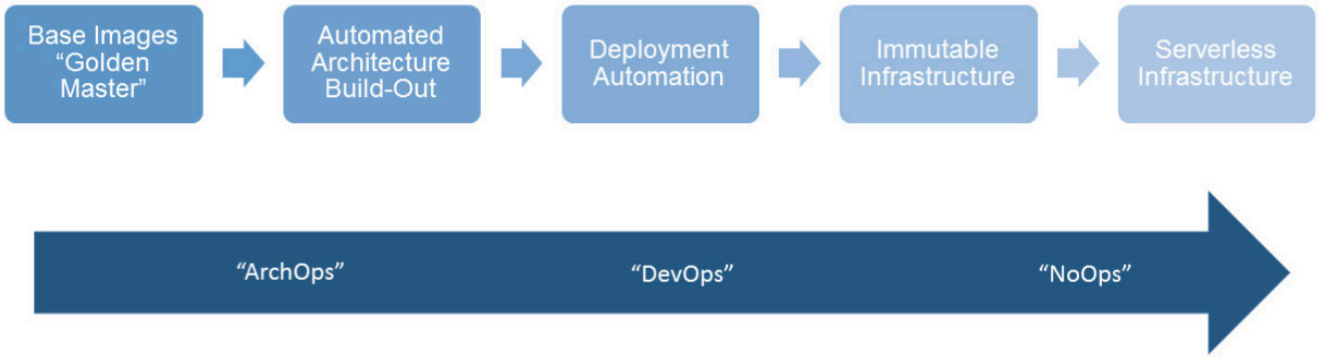
When security professionals embrace this approach, they have far greater impact than in the past. This is actually an opportunity for security professionals to get what they have always dreamed of: introducing security earlier in the development process. Rather than retroactively enforcing security policies — and always being behind — they are part of the architecture planning process from Day 1. They can code their desired specifications into templates, and always know that their desired configurations are enforced. They no longer need to be consulted on each and every infrastructure change, they only need to be consulted when the infrastructure templates change in a significant way. This means less repetitive busy-work and more focus on real issues.

The Future of DevOps on AWS

The world's largest enterprises now host complex, mission-critical infrastructure on IaaS platforms like Amazon Web Services. You can maximize the power of AWS when the infrastructure is managed by a highly integrated DevOps team that is committed to process and communication.

There was a time when a system administrator would show the uptime on his/her systems with pride. DevOps teams take equal pride in high availability, but also in KPIs like deployment frequency, lead time from ticket to fully-configured infrastructure, and mean time to recovery (MTTR). In this new universe, better outcomes do not happen because a team throws money at more hardware or software vendors; it happens when IT operations teams spend time automating infrastructure and deployment (or hire a company like RapidScale to do it for you).

The pressure to deliver software applications quickly continues to increase. This is part of why automation has always been a crucial component of DevOps philosophy. But fully automated environments require a sophisticated development process, and in the rush to production, that can add unwanted complexity. This is why organizations need to be strategic about selecting the aspects of DevOps the team can currently handle. You do not need to implement every DevOps best practice in order to enjoy the benefits of DevOps principles. DevOps is a spectrum, from most basic to advanced:



Wherever a team is on the DevOps spectrum, any degree of automation can lead to a gain in efficiency that directly translates into cost-efficiencies. A team that is less hampered by provisioning timelines, lengthy deployment approval processes, etc. will be able to concentrate on improving the organization's applications, not on repetitive manual work. IT will no longer be an organizational cost, but a driver of innovation, powering new lines of business and working more closely with business leaders to meet organizational goals. Although some teams may resist this dramatic organizational shift, the inexorable progression of the industry will ultimately yield leaner, more agile teams than were possible in IT just five years ago.

About RapidScale

RapidScale, the leader in compliant cloud solutions, provides end-to-end cloud strategy, RapidScale helps customers migrate, run, and operate mission-critical workloads on AWS with security, scalability, and efficiency baked in. RapidScale's Cloud Reliability Platform combines world-class engineering talent, policy-as-code, and integrated tooling to enable customers to confidently meet compliance regulations, security requirements, cost control, and high availability. Together with our team of dedicated certified engineers and decades of IT management experience, we ensure our customers' success across every stage of the Cloud Adoption Framework and governance.

- Solution Provider
- Managed Services Provider
- SaaS Services Competency
- DevOps Services Competency
- Security Services Competency
- Generative AI Services Competency
- L1 MSSP Services Competency
- Migration Services Competency
- Healthcare Services Competency
- Small & Medium Business Services Competency