ThoughtWorks®

PERSPECTIVES

Rethinking Cloud Development

Part 1: Architecture and Devops

Exploring new approaches and their adaptations from real-world projects

Share this ebook.

Contents

Rethinking the Way we Build on the Cloud

To say that cloud computing has revolutionized the industry, is an understatement. From changing revenue models and overhauling operations to fueling innovation, the cloud has driven massive changes. Isn't it time for some shakeups at the heart of it all - the way we build on the cloud?

Our collection of essays, the first in a series, explores some of the recent evolution in thought on designing and developing applications for the cloud. Part I focuses on architecting and devops on the cloud. It includes aspects such as setting up environments, immutable servers, blue-green deployments on AWS and configuration drift. Read on...

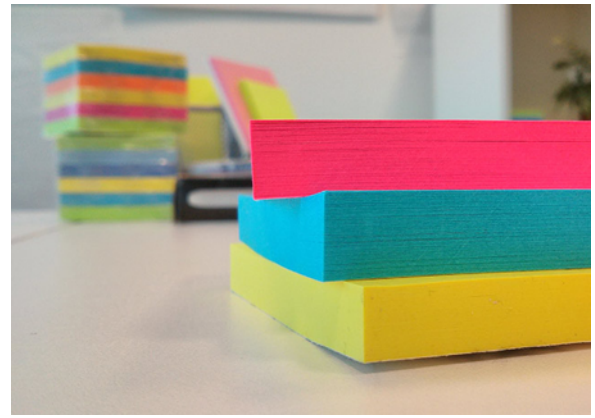
Architecting Applications on the Cloud

“Layering the Cloud”

Make the most of the cloud by splitting its architecture

Peter, **Developer**

One of the great things about the cloud is the way you can just run a bit of code or a bash script, and before a Windows admin can open their GUI, you've got a running box. This opens up a host of opportunities and new patterns. Martin Fowler recently posted about the Phoenix Server pattern where machines are simply thrown away rather than upgraded. However, this requires you to look at the way you architect infrastructure differently.



To help, you can split your cloud architecture into three layers: Visible, Volatile and Persistent.

Visible: This is the layer between the cloud and the rest of the world. It includes public DNS entries, load balancers, VPN connections, etc., which are fairly static and consistent. If you have a website, you will want the same *A Name* records pointing at your static IP. Things in this layer rarely change, if they do, they are for very deliberate reasons.

Volatile: This is where the majority of your servers are. The known state (number of servers, what versions of which software they run etc.) changes frequently, perhaps even several times per hour.

Even in a traditional datacenter, your servers are upgraded, security patches are applied, new versions of software deployed, extra servers added, etc. On the cloud this is even more volatile when you use patterns such as the Phoenix server and machines are rebuilt with new IP addresses. You should be able to destroy the entire volatile layer and rebuild it from scratch without incurring any data loss.

Persistent

This is where the important stuff is kept, that you can't afford to lose. Ideally only the minimum infrastructure to guarantee your persisted state is here. For example, the DB server itself would be in the volatile layer, but the actual state of the DB server, its transaction files, etc. would be kept on some sort of robust 'permanent' storage.

In Conclusion

By organizing your infrastructure around these three layers you are able to apply different qualities to each layer. For example, the persistent layer would require large investment into things like backup and redundancy to protect it whilst this is completely unnecessary for the volatile layer. Instead the volatile layer should be able to accommodate high rates of change while you will want to maintain a considerably more conservative attitude towards the persistent and visible layers.

"Configuration Drift"

Ways to avoid configuration drift

Kief, **Continuous Delivery Practice Lead for Europe**

Cloud hosting makes it easy and quick to provision new servers from a template image. This ensures all of the machines in your infrastructure, whether for development, testing, or production, are consistent when they are created. Over time however, running servers tend to diverge until they can become quite different from one another. This phenomenon is called *Configuration Drift*.

There are a number of ways this can happen. People may make ad-hoc, manual changes to servers to fix problems, try out new things, or make necessary updates. In some cases, applications make changes to configuration or data as part of their normal runtime operation. If nothing else, newly created server images may have newer versions of system and application packages, for example security patches. These new servers will not be quite the same as those created before, and in time these differences may be different enough to affect consistent operations.

Dealing with Configuration Drift

There are two main methods to combat configuration drift:

1. Use automated configuration tools such as Puppet, Chef, or Ansible, and run them frequently and repeatedly to keep machines in line.
2. Rebuild machine instances frequently, so that they don't have much time to drift from the baseline, following the Phoenix Server or the Immutable Server pattern.

The challenge with automated configuration tools is that they only manage a subset of a machine's state. Writing and maintaining manifests/recipes/scripts is time consuming, so most teams tend to focus their efforts on automating the most important areas of the system, leaving fairly large gaps.

There are diminishing returns for trying to close these gaps, where you end up spending inordinate amounts of effort to nail down parts of the system that don't change very often, and don't matter very much day to day.

On the other hand, if you rebuild machines frequently enough, you don't need to worry about running configuration updates after provisioning happens.

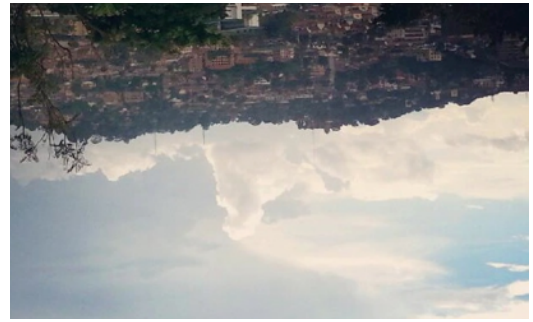
However, this may increase the burden of fairly trivial changes, such as tweaking a web server configuration.

“Environments on the Cloud and Immutable Servers”

Principles to guide your approach to environments on the cloud

Ben, **Developer**

The newly launched Mingle SaaS runs entirely on the AWS cloud. As there was no existing system to modify or integrate with, we could design the architecture from scratch. This led us to rethink the role of environments in our development and deployment process.



What’s wrong with the traditional approach to environments?

In traditional data-center-based applications, there are usually a small, fixed number of environments into which the application is deployed (production, staging, test, development, etc.). The availability and nature of these environments is strongly constrained by the availability of hardware and infrastructural systems for them to run on.

We would like all our environments to be as similar to production as possible, but physical hardware and traditional infrastructural systems like databases are expensive and slow to provision. In practice, this means that there is a sliding scale of realism in the environments as you go from development to production. Development environments tend to be smaller than production (load-balanced applications may have only one server, when there are a dozen in production), they often use alternatives for some components (different database servers) and they frequently share components (fileservers, databases, monitoring systems) when these are dedicated in production. Similarly for test and even staging environments; though they may be more realistic the closer they are to production.

This *variation* between environments causes several problems. Most obviously, some bugs are found further down the pipeline, rather than when developers are working on the code. This increases the cost of fixing the bugs. Another problem is that supporting the variation in environments increases the complexity of the code. And, more subtly, we end up making decisions which cause our architectures not to be optimized for the real deployment environment, because developers are divorced from the reality of running the system in that environment.

The inevitable restriction on the *number* of environments also causes problems. Availability of environments can cause delays or influence us to miss out on the testing that we would like to do. Maintenance of the environments, like cleaning up after stress tests, also takes time.

How have we approached environments in the cloud?

We have found that building a system that runs entirely on the cloud has enabled us to reconsider our use of environments and ensure that we don't fall foul of any of these problems.

(continued...)

“Environments on the Cloud and Immutable Servers”

Principles to guide your approach to environments on the cloud

(...continued)

Following the three principles described below has increased the reliability of our system, streamlined our deployment process and sped up our development.

A. Ad hoc environments

Ensure that environments can be created and destroyed quickly and on demand.

We ensure that it is trivial to deploy a complete, new instance of the system, rather than use a set of pre-existing environments or run partial/crippled environments. For example simplified systems deployed on developers' boxes.

This means that we use realistic environments when we test, so we are less likely to encounter problems further down the pipeline. It means that we can create short-lived environments for special purposes. It means that there is no contention for the use of a small set of realistic environments. And it means that we can dispose of environments when we are done with them, rather than have them accumulating configuration drift.

To achieve this, we automate everything required to create a new environment. A script takes the name of the environment as a parameter and creates everything that the environment needs, from scratch. Another script destroys the environment when we're done with it. In practice, we create many completely new development environments several times a day. We also have two long-running environments: for staging changes before release and to run our production system.

B. Shared-nothing environments

Do not share any infrastructure between environments.

Every environment has its own database servers, package repositories, monitoring systems and so on. There are three reasons for this:

1. Shared infrastructure creates a touch point that allows environments to interfere with one another. For example, heavy load on one environment could affect the performance of another one.
2. It causes migration problems. We would like every infrastructure change we make to be tested as it progresses up a delivery pipeline. If two environments in the pipeline share a piece of infrastructure then an upgrade will affect both of them at once.
3. Lack of sharing simplifies the versioning of packages deployed to the system; with no shared package repository, it is clear what version is deployed because there is only one version present in any environment.

This approach does have some costs. We have to pay for the infrastructure for

(continued...)

"Environments on the Cloud and Immutable Servers"

Principles to guide your approach to environments on the cloud

(...continued)

every environment (e.g. an Relational Database Service (RDS) instance). And creating everything whenever we spin up an environment can take time (again, an RDS instance).

C. Cookie-cutter environments

Minimize the difference between environments.

Every environment we create is essentially identical. In particular, network topologies, instance types and clustering approaches are the same in both development and production environments.

Configuration required for individual environments is limited to the environment name, the root domain for the environment's Domain Name System (DNS) entries, an email address for alerts and some security details. Because there are no significant differences between environments, our deployment process is relatively simple and we see very few bugs caused by differences between the environments. We can do this because we don't share any infrastructure between environments, so there is no need to provide configuration to identify infrastructural services.

Auto-scaling also minimizes differences. All environments start equal at deploy-time and then adapt to the load placed on them, rather than having heavily-loaded environments like production provisioned with more resources than, say, a development environment.

Principles in practice

Following these principles has required us to be very disciplined when making changes and forced us toreject some otherwise "attractive" approaches. However it has been worth it. Keeping all environments completely isolated prevents contamination. Also, all changes to all components in our system are tested by going through the pipeline rather than being imposed on multiple environments at once.

Ensuring that we can spin up such environments quickly, as needed, means that we never waste time waiting for environments to be available or reconfiguring them to our immediate needs. Using full environments, identical to production, at every stage in our development pipeline means that we don't experience errors caused by unanticipated differences between environments. Also, fully integrating all the system components early in the pipeline, gives us fast feedback on inconsistencies.

Immutable Servers

We treat all our virtual servers as immutable. When we upgrade our system we create brand new servers and destroy the old ones, rather than upgrading them. This is a logical extension of the phoenix server approach in which servers are regularly recreated from scratch.

(continued...)

"Environments on the Cloud and Immutable Servers"

Principles to guide your approach to environments on the cloud

(...continued)

Why immutable servers?

They allow us to be absolutely certain of the state of a provisioned server (more reliably so than declarative configuration management tools). It's impractical to specify every single detail of a system, so the end state that it reaches is inevitably dependent on what has gone before. For instance, consider how tough it is to remove a previously installed package. If you simply start again from scratch, you can be certain that there's nothing installed, but what you need.

Immutable servers also simplify our system. The deployment infrastructure is simplified because it doesn't have to support different cases for upgrades and new environments (similar to ad-hoc environments). Testing is simpler because we don't need different tests for upgrading servers and creating them from scratch. And it helps us fix deployment problems by rolling-forward because any problematic servers will be destroyed.

The decisions we make in order to support immutable servers in our system also have a simplifying and clarifying effect on the architecture. They often help (or force) us into doing the right thing and have unanticipated benefits later on.

The implications of using immutable servers

System upgrades are somewhat slower because creating virtual servers from scratch takes a bit of extra time. Also, any change to a server requires a redeploy as there is no mechanism for modifying them in place. As servers are recreated from scratch, any data that changes between deploys will be lost. This needs to be factored when designing the system architecture, most obviously for application data. We deal with this by separating our architecture into the three layers as detailed earlier. Ensuring that important data is in the persistent layer simplifies event-sourcing, with the events stored in the persistent layer and replayed on deployment so applications can recreate the state they need. It is important to clearly separate persistent from volatile data and ensure that you are persisting only the data that you need to. This clarifies application design, ensures that you are not unnecessarily paying the cost of persisting data and simplifies the process of discarding outdated volatile data when you deploy.

Logs also need to be preserved across deployments so that we can retrospectively investigate problems without worrying about an intervening deployment. The best way to do this is to ship all logs to a central server (e.g. rsyslog), that treats them as application data and stores them in the persistent layer.

Finally, having immutable servers means that you can't update system configuration (like server addresses), without redeploying. If server addresses change across deployments, you end up with dependency problems, where you have to update them in a certain order, and possibly unresolvable circular dependencies. The solution is to decouple addresses from the physical servers. Our preference for this is to use Domain Name System (DNS).

“Implementing Blue-Green Deployments with AWS”

Reduce deployment risk

Danilo, **Lead Consultant**

An important technique for reducing the risk of deployments is known as Blue-Green Deployments. If we call the current live production environment “blue”, the technique consists of bringing up a parallel “green” environment with the new version of the software and once everything is tested and ready to go live, you simply switch all user traffic to the “green” environment, leaving the “blue” environment idle. When deploying to the cloud, it is common to then discard the idle environment if there is no need for rollbacks, especially when using immutable servers.

If you are using Amazon Web Services (AWS), there are a few options to implement blue-green deployments based on your system’s architecture. As this technique relies on performing a single switch from “blue” to “green”, your choice will depend on how you are serving content in your infrastructure’s front-end.

1. Single Amazon Elastic Compute Cloud (EC2) instance with Elastic Internet Protocol (IP)

In the simplest scenario, all your public traffic is being served from a single EC2 instance. Every instance in AWS is assigned two IP addresses at launch -- a private IP that is not reachable from the Internet, and a public IP that is. However, if you terminate your instance or if a failure occurs, those IP addresses are released and you will not be able to get them back.

An Elastic IP is a static IP address allocated to your AWS account that you can assign as the public IP for any EC2 instance you own. You can also reassign it to another instance on demand, with a simple API call. In our case, Elastic IPs are the simplest way to implement the blue-green switch. Launch a new EC2 instance, configure it, deploy the new version of your system, test it, and when it is ready for production, simply reassign the Elastic IP from the old instance to the new one. The switch will be transparent to your users and traffic will be redirected almost immediately to the new instance.

2. Multiple EC2 instances behind an Elastic Load Balancer (ELB)

If you are serving content through a load balancer, then the same technique would not work because you cannot associate Elastic IPs to ELBs. In this scenario, the current blue environment is a pool of EC2 instances and the load balancer will route requests to any healthy instance in the pool. To perform the blue-green switch behind the same load balancer you need to replace the entire pool with a new set of EC2 instances containing the new version of the software. There are two ways to do this: automating a series of API calls or using AutoScaling groups.

Every AWS service has an API and a command-line client that you can use to control your infrastructure. The ELB API allows you to register and de-register EC2 instances, which will either add or remove them from the pool. Performing the blue-green switch with API calls will require you to register the new "green" instances while de-registering the "blue"

(continued...)

“Implementing Blue-Green Deployments with AWS”

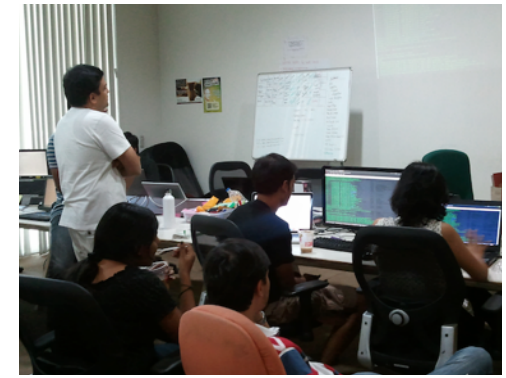
Reduce deployment risk

(...continued)

instances. You can even perform these calls in parallel to switch faster. However, the switch will not be immediate because there is a delay between registering an instance to an ELB and the ELB starting to route requests to it. This is because the ELB only routes requests to healthy instances and it has to perform a few checks before considering new instances as healthy.

The other option is to use AutoScaling. This allows you to define automatic rules for triggering scaling events; either increasing or decreasing the number of EC2 instances in your fleet. To use it, you first need to define a launch configuration that specifies how to create new instances: the Amazon Machine Image (AMI) to use, instance type, security group, user data script, etc. Then you can use this launch configuration to create an auto-scaling group defining the number of instances you want to have in your group. AutoScaling will then launch the desired number of instances and continuously monitor the group. If an instance becomes unhealthy or if a threshold is crossed, it will add instances to the group to replace the unhealthy ones or to scale up/down based on demand. AutoScaling groups can also be associated with an ELB and it will take care of registering and de-registering EC2 instances to the load balancer any time an automatic scaling event occurs. However the association can only be done when the group is first created and not after it is running. We can use this to implement the blue-green switch, but it requires a few non-intuitive steps:

1. Create the launch configuration for the new “green” version of your software.
2. Create a new “green” AutoScaling group using the launch configuration from step 1 and associate it with the same ELB that is serving the “blue” instances. Wait for the new instances to become healthy and get registered.
3. Update the “blue” group and set the desired number of instances to zero. Wait for the old instances to be terminated.
4. Delete the “blue” AutoScaling group and launch configuration.



This procedure will maintain the same ELB while replacing the EC2 instances and AutoScaling group behind it. The main drawback to this approach is the delay. You have to wait for the new instances to launch, for the AutoScaling group to consider them healthy, for the ELB to consider them healthy, and then for the old instances to terminate. While the switch is happening there is a period of time when the ELB is routing requests to both “green” and “blue” instances which could have an undesirable

(continued...)

"Implementing Blue-Green Deployments with AWS"

Reduce deployment risk

(...continued)

effect for your users. I would instead consider Domain Name System (DNS) redirection.

3. DNS redirection using Route53

Instead of exposing Elastic IP addresses or long ELB hostnames to users, you can have a domain name for all your public-facing URLs. Outside AWS, you could perform the blue-green switch by changing CNAME records in DNS. In AWS, you can use Route53 to create a hosted zone and define resource record sets to tell the DNS how traffic is routed for that domain. Bring up a new "green" environment (a single EC2 instance, or ELB), and then update the resource record set to point the domain/subdomain to the new instance or ELB.

There is also a better alternative. Route53 has an AWS-specific extension to DNS that integrates better with other AWS services, and is cheaper too -- alias resource record sets. They work pretty much the same way, but instead of pointing to any IP address or DNS record, they point to a specific AWS resource: a CloudFront distribution, an ELB, an S3 bucket serving a static website, or another Route53 resource record set in the same hosted zone.

Finally, another way is by using Weighted Round-Robin. This works for both regular resource record sets as well as alias resource record sets. You have to associate multiple answers for the same domain/sub-domain and assign a weight between 0-255 to each entry. When processing a DNS query, Route53 will select one answer using a probability calculated based on those weights. To perform the blue-green switch you need to have an existing entry for the current "blue" environment with weight 255 and a new entry for the "green" environment with weight 0. Then, simply swap those weights to redirect traffic from blue to green. The only disadvantage is that propagating DNS changes can take time, so you would have no control over when the user will perceive it. The benefits are that you expose human-friendly URLs to your users, the switch happens with near zero-downtime, you can test the new "green" environment before promoting it, and you can do canary releases for free.

4. Environment swap with Elastic Beanstalk

Elastic Beanstalk, Amazon's platform-as-a-service offering supports Java, .Net, Python, Ruby, NodeJS and PHP. It has a built-in concept of an environment that allows you to run multiple versions of your application, as well as the ability to perform zero-downtime releases. Therefore, the blue-green switch simply consists of creating a new "green" environment and following the steps in the documentation to perform the swap.

Conclusion

Blue-Green deployment is an important technique to enable Continuous Delivery, especially on the cloud. It reduces risk by allowing testing prior to the release of a new version, while enabling near zero-downtime deployments, and a fast rollback mechanism. Cloud providers such as AWS enable you to easily create new environments on-demand and provide different options to implement Blue-Green deployments.

"Cloud Based DevOps: Possible on Windows?"

Ways to overcome challenges with Devops on the cloud

Rachel, **Lead Consultant**

As a developer in the Windows environment for over 10 years, I have overcome my fair share of challenges. I was developing before Google was useful, when finding information required an MSDN license and when deployment meant copying a DLL onto a file-share, or worse, installing it with a CD.

The world of development and operations tooling and practices has moved on considerably since then, thankfully, as it was wrought with fear that at any moment something could go horribly wrong in production! And then you would have an angry operations colleague on your hands.

Thankfully now we are making friends. The movement of DevOps allowed us to break down the silo, to work together despite our different concerns. I'm trying to get more features into an environment and my ops colleague is trying to keep production stable. We've realized that by working together we can release lots of small well-tested features into lots of production-like environments and when we are happy they won't break, we can release them together with little fear, and skip merrily along into the sunset. Or at least that's how it appears when you are looking at the unix land from the dark underworld of Windows.

"Tooling is a solved problem in DevOps. Except if you are in Windows"

I keep hearing "tooling is a solved problem in DevOps". But is that really the case on Windows? Let's look at some of the challenges you will face when trying to build infrastructure and deploy applications on the cloud on a Windows platform. What can we as a community do to make it a better and brighter place? Maybe then, we can be the ones doing the skipping.

With any problem I face I like to break it up into understandable pieces that I can tackle one by one. So, let's break up the challenge of DevOps for the cloud into three parts: Build, Deployment and Environment Provisioning.

1. Build:

This is one of the few areas that is actually a solved problem in Windows. The real challenges are self-enforced through bad tooling choices. Build is really made up of source control management (SCM), orchestration of compilation, testing and packaging your application for deployment. For this you should choose tools that support the practice of continuous integration, allow you to manage your builds exactly how you want them, visualize at what stage your build is at and give the team immediate feedback if it is broken.

2. Deployment:

This is where it gets a little harder. It should be as easy as taking your packaged application and dropping it onto a server somewhere. In Windows

(continued...)

"Cloud Based DevOps: Possible on Windows?"

Ways to overcome challenges with Devops on the cloud

(...continued)

you might want to configure and restart IIS, install and run your services or any other 3rd party software and services, run your database scripts and run a smoke test to check it is all functioning together as expected. Taking your packages and copying them on the server is actually quite easy if you are using a packaging mechanism like MSI, for example. These tools can help with it:

- **PowerShell:** PowerShell has the ability to copy over your files, restart IIS, start your services and many other remote commands available on your server - given your server supports the remoting command you need.
- **Octopus:** Octopus is a new tool that installs tentacles onto your servers, setting up a remote connection. If you package your application with Octopak, it can also copy over your files and start IIS and other services.

Pain

This all sounds pretty easy and awesome, but it starts to get painful really quickly as soon as you want to deploy something other than your application or components. Something that someone else wrote that your application depends upon, which, let's face it, is a very common requirement. First of all, there isn't a common packaging mechanism in Windows. People use a combination of msi's, ini's, Nullsoft, Innosetup and others that I am sure you will discover. Not all of those can be automated or accessed through the command line and often require a human to click a button to install. GUI installs are really hard, if not impossible to automate. Some might provide command line scripted installs, but they offer less functionality than their GUI counterparts (e.g. NuGet). So what can you do?

a. **Choose tools you can script and automate**
   Unfortunately that might be out of the question. The software may have already been purchased or may not exist. But if you do have the opportunity, make the automation and testing of 3rd party tools a first-class concern, so that you can repeatedly deploy, configure and test it without worrying about introducing any opportunity for human error.

   Something Chocolately: There is some good news though. Let me introduce you to Chocolatey. A package install tool such as the likes of Homebrew on Mac, which lets you install system packages. It uses PowerShell scripts and Nuget to install system applications for you. It could be the start of a standard packaging mechanism on the Windows platform, and for that reason it is worth keeping an eye on.

b. **Build it yourself, but better:** If you can't buy it, build it. But build it with a common packaging mechanism like an msi and the ability to deploy and configure it in a script. Or leverage an open source project and add features you need by contributing. Again, that might also be out of the question, as it could be too expensive and time-consuming. There is a reason you had to buy 3rd party software or tool in the first place. In reality, for the time being you'll probably have to accept that it can't be automated. But it isn't all doom and gloom.

(continued...)

"Cloud Based DevOps: Possible on Windows?"

Ways to overcome challenges with Devops on the cloud

(...continued)

c. Create the need for change: Much of the reason software on Windows doesn't have a common packaging mechanism or uses a GUI to install, is due to customer requirements. Once upon a time we didn't care or know to automate our deployments and we were content with just deploying them through the GUI one time only. So the vendors and toolmakers met that need. However, that need has changed, and only with the push from customers will the tide turn.

3.Environment Provisioning

And now we're onto the really tough problem. In the world of the cloud you have a couple of options, you can host your own cloud or use a third-party hosted solution. Regardless of what you choose the benefit you gain is Virtualization. Virtualization means you won't be building all your environments from scratch on one box. Through shared use of server resources you can create lots of production-like environments at low cost, which will allow you to do parallel testing and scale your infrastructure.

Most hosted cloud options provided by vendors like Azure from MS, EC2 from Amazon, provide Infrastructure as a Service (IaaS). This will give you some basic Windows infrastructure, like your operating system, but that is only part of the problem solved. Most real systems are heterogeneous, which means they need to be configured for specific purposes. Examples of the types of environments you might need are development, continuous integration (CI) and production-like. Development is likely to have

development tools and stubs to external components and applications. CI may only include your build tool and basic frameworks like .NET, but won't need development tools or real external components and applications. Production-like environments will have real external components and systems, but won't need build and development tools. You could start with a base image and then use configuration management tools like Puppet and Chef to configure your environment to your needs. Alternatively, you can build the entire environment from scratch using Puppet and Chef. But hold on, I thought we were on Windows?

Configuration Management

We are on Windows, and fortunately massive improvements have been made to configuration management tools like Puppet and Chef. They now have a lot of built in support for Windows packages and configuration. A large part of the environment configuration can be performed using these tools and for everything else there are always executable blocks, that allow you to call out to PowerShell. If you do use an executable block, ensure you replace it with the actual package should it start being supported by default. It isn't clean or easy to manage lots of executable blocks.

(continued...)

"Cloud Based DevOps: Possible on Windows?"

Ways to overcome challenges with Devops on the cloud

(...continued)

PowerShell & WMI

For everything else there is PowerShell. Specifically, WindowsFeatures and PowerShell for Windows Server 2008 r2 has a server manager module. This module has comandlets that allow you to add a Windows server feature. So basically with PowerShell and WMI (Windows Management Infrastructure and WinRM (Windows Remoting)) you can pretty much do anything that the Windows Server API will let you. Including automating Active Directory commands. The caveat is that your Windows Server needs to be at least Windows Server 2008, when PowerShell support became a built-in feature. Before then... good luck!

And lets not forget our new friend Chocolatey, which will allow you to install system packages. More and more applications are becoming available in Chocolatey.

However, it isn't all sunshine and lollipops. WinRM is actually pretty painful and fiddly to use and PowerShell is an ugly and procedural language. You have to be careful not to let it grow arms and legs and become difficult to understand, and therefore, difficult to maintain. We all know what happens to that kind of code.

More Pain

There are a few other pain points that I would be remiss not to mention. Let's start with registries. In Windows we have 32bit and 64bit registries. Both Puppet and Chef have issues with living on one registry and installing to another. Once you end up in this space be prepared to debug and perhaps jump through some hoops to get things working.

Other irritating "features" you need to manage are Windows updates and ISO mounting. ISO mounting is still not built into the Windows operating system, so you'll need to download something like Virtual Clone Drive.

And finally there is the cost. Even on Azure, Windows environments are more expensive than Linux, but good luck using that as an excuse to port all your software.

So now that I have thoroughly depressed you and perhaps made you consider just doing it on Linux. Let's talk about the light at the end of the tunnel. Each of these problems can currently be solved by one of the following solutions:

✓Manage the problem

Being aware of a problem is the first step to fixing it. Don't go into Windows automation believing it's going to be easy and you won't have to deal with these little niggles. You will. Therefore the best defense is an is an offense. Be prepared, be careful with registries, ensure you manage Windows updates so they don't manage you, and use a version of a Windows that

(continued...)

"Cloud Based DevOps: Possible on Windows?"

Ways to overcome challenges with Devops on the cloud

(...continued)

supports PowerShell and WinRM. Right now, I recommend Windows Server 2008 and above.

✓Create the need for change

I've said it before but it is really the only way that things will get better. Microsoft and vendors have a responsibility to respond to requirements from their customers. So create those requirements by making them visible at scale. If we all started trying to do DevOps on Windows, the vendors would respond by trying to make it easier for us. We just need to remind them that using a GUI is not the kind of ease we are after. Push the community and support open source software (OSS). Even Microsoft is supporting OSS now. For example, they have open sourced MVC and their Entity Framework. Demonstrating that a lot of fantastic tooling and innovation can be built and trusted in this space.

Automate what you can, accept what you can't

Given the nature of software, I'm sure if I write this again in five years, we won't have these problems and we'll have awesome tools for automating the creation of our infrastructure and the deployment of our applications.

But right now we have to create the requirement for change and the only way to do that is for everyone to try and automate what they can. Use the best tool there is and petition to have it better.

Automate what you, and accept what you can't... for now.

"Evolution of Mingle to SaaS"

A case study of how we moved Mingle to the cloud

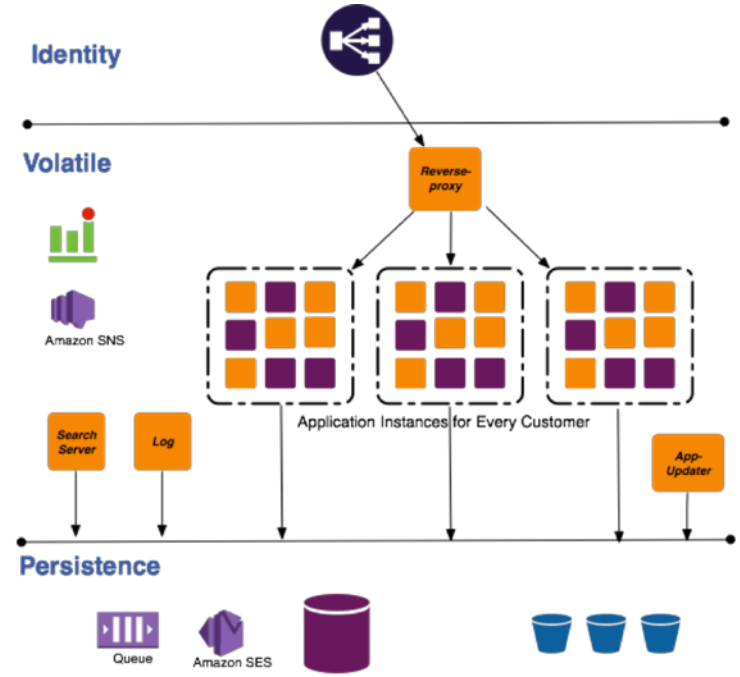
Sudhindra, **Developer**

Mingle on the cloud. It all started back in 2012

On our journey to allow for faster delivery times, wowing customers with new features and allowing for quicker support, we made a decision to move Mingle to a 'Software as a Service' on the cloud. We spent a few months architecting Mingle to suit the features and limitations of the cloud and this is the story about that effort.

Our first challenge was to build something quick and customer-facing. We wanted to engage with the customer as early as we could by providing a cloud-based web trial. Architectural changes were introduced piece-by-piece to allow for assessment and tweaking. The trial system was the same Mingle available as a packaged installer - except that we were doing the operations work. The user could thus have a dedicated instance of Mingle on the cloud and get started on it, with just a sign-up.

While doing this we also wanted to be cost-conscious. Considering how costs on the cloud add up over time, we were constrained by how much we could spend on each customer. Given how most cloud services operate and charge, the majority of our costs were using specialized services like compute, database and network traffic. Cost of storage, load balancing and stock services like caching turned out to be a percentage of the specialized and more expensive services. In order to minimize these costs, we examined the option of building such a system, and found examples on Unix systems. On Unix, you can build 'jailed environments' on top of *chroot* or its



Linux clone *lxc*. We could also get control on how these services could be expanded or contracted based on metrics of usage. This was our first architecture change.

Using *lxcs* as our control and isolation mechanism worked well since:

- *lxc* were guaranteed by Linux to be isolated and independent environments.
- We could multiply the *lxc* or throw it away.
- A simple lookup table would keep track of allocated *lxcs* for customers.
- This allowed us to test on the cloud without building real multi-tenancy.
- It allowed us to efficiently use the resources of EC2 compute instances.

We used this mechanism for a while, along with a

(continued...)

"Evolution of Mingle to SaaS"

A case study of how we moved Mingle to the cloud

(...continued)

reverse-proxy for routing requests correctly to the servers. As our customer base started scaling, we faced constraints when managing such growth with the *lxc* architecture.

Our costs started growing, almost linearly with requests, which wasn't sustainable. Running a service on SaaS can be profitable only if you can control your costs aggressively, and spend money only when there is substantial load on your system. With the *lxc* architecture there was a base cost of running each instance, as the lxc continues to run on the instance CPU even when there are no customer requests for the instance. Given how *lxc* works, we were restricted in turning them down, as they continued to be running processes and occupied some minimum CPU time. Since we had dedicated *lxc* instances for each customer, as the customer base grew, we were adding more compute instances to support them. This posed another problem - updating and restarting each *lxc* container on a Mingle feature update. This method would be prone to errors, be a time sink, and would also incur downtime, adversely affecting customer satisfaction.

We also had to build additional adapters to fit the *lxc* based architecture in the cloud:

- *Reverse-proxy* for routing requests for a particular customer to an appropriate lxc.
- *apt-repo*- a service to store and update all the instances with all the packages that include and support Mingle - Mingle server, sun jdk, linux updates, and so on.
- Our own adapters to use AWS services.

Mingle on the cloud - new and improved

We needed a better way to do things and reduce our costs while being able to scale. We started examining our approach and comparing it with technologies being launched by Amazon on AWS.

Our analysis revealed that the Mingle load currently does not exceed 30req/minute and would remain at this level for the near future. This built a clear case for multi-tenancy, as that would allow us to scale based on number of active requests (load) as opposed to provisioned customer instances. It would keep the cost constant - we could serve a large number of customers with the minimum number of compute instances - while allowing for data security and efficient usage of resources.

We have since built in multi-tenancy in Mingle's architecture. It leverages multi-tenancy at every request and reconfigures the application for that request based on the information about the customer. Customer's data is still isolated at the persistence level while the requests are multiplexed at the application level.

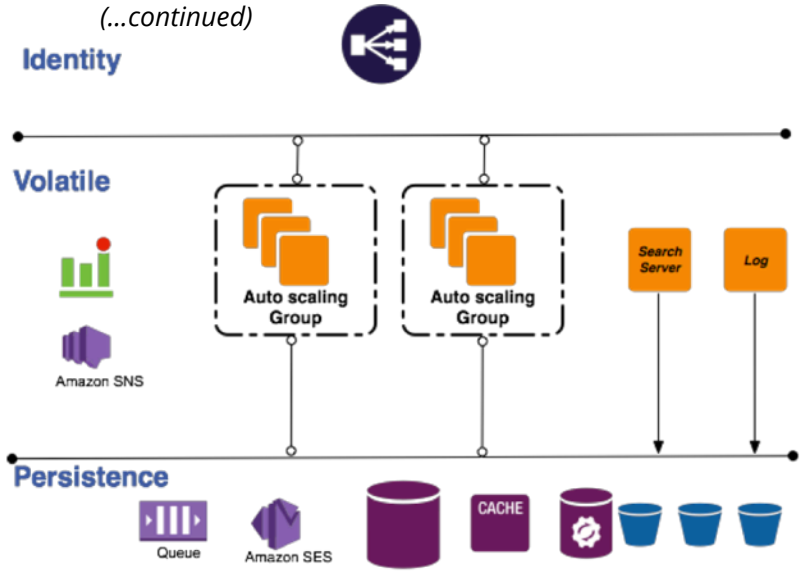
Once we implemented multi-tenancy, we were able to reduce the number of compute instances to two (to ensure redundancy and distribute load) for all the web requests we needed to handle. Having such an architecture allowed us to move the application state to persistence and identity layers, while keeping the volatile resources ignorant to state.

(continued...)

"Evolution of Mingle to SaaS"

A case study of how we moved Mingle to the cloud

(...continued)



Introduction to Amazon Cloud Formation (CF)

While re-architecting, we were introduced to CF, which promised to relieve us from the pain of writing deployment code using APIs. It also ensured that we would not have to deal with 'eventual consistency' issues with provisioning individual services.

We would have to change the way we thought about our application though. Some of our deployment code would have to be removed or repurposed to suit CF 's requirements.

CF allows you to define your resources and their dependencies and then proceeds to correctly provision them. CF also brought with it a set of features we were looking to build in -- autoscaling, high availability, distributing for redundancy across zones/regions, zero downtime and failsafe updates at no cost to the customer.

Amazon Machine Image(AMI)

AMIs are the recommended way of provisioning resources with CF, and a way to support continuous deployment with CF, as they ensure traceability and are packaged in a way that CF can consume. Since each AMI is identified by a unique id, we use it to tag our latest AMIs for the next deployment. All these updates are stored on Amazon Simple Storage Service (S3) that CF has access to. Our deployment process now follows a simple pattern where CF creates a stack for the new version of the application and retires the old one. This provides us with zero downtime for our web-facing server. Having configured CF stacks in such a way that the autoscaling group spans across multiple zones, we achieve automatic high availability by relying on the AWS system to do the right thing in case an application instance fails.

There are times when we might face errors during provisioning, resulting in failures. However, these errors are hidden from the end user as they are not exposed to the failed stack. The original stack continues to run and serve requests as if nothing has happened. We are notified of such failures and can do our due diligence to fix the deployment.

What's next?

As we continue to improve our codebase and deployment process, we have been building our new features using ElasticBeanstalk. It is a tool to provision services (which can include stacks with multiple resources) with a better alignment to continuous delivery. Stay tuned for more on ElasticBeanstalk and about a tool that we built to support continuous delivery on AWS.

Stay tuned for Part 2: Cloud-based Testing, XD and a lot more...

Peter Gillard-Moss

I'm a systems developer at Thoughtworks and have been working in IT since 2000 for a variety of domains and team sizes, using a wide range of technologies and languages -- from Windows, .NET, Java to Linux, Ruby and Clojure.

I like my culture and I'm a terrible high-brow snob when it comes to music and literature. When I'm not being pompous I'm cycling or running stupidly long distances.

I am a developer and have spent the last ten years looking for ways to avoid unnecessary work. When all else fails, I like to write code in languages that haven't been designed to hurt me.

This week my interests include cloud computing, functional programming and renaissance theatre.

Ben Butler-Cole

I'm a Lead Consultant at ThoughtWorks. I have 10 years of experience in systems development, having worked on a wide range of technologies and the integration of many disparate systems. At ThoughtWorks, I've coached teams on Agile and Continuous Delivery technical practices and have played the role of coach, trainer, technical lead, architect, and developer. I am also a member of the Technical Advisory Board to the CTO, which regularly produces the ThoughtWorks Technology Radar.

I'm fascinated by problem solving and has discovered that people problems are often more difficult to solve than software ones.

Rachel Laycock

Danilo Sato

I'm a Lead Consultant at ThoughtWorks. I enjoy using software to solve hard problems, and helping people improve their cycle time between having an idea and having software deployed to production.

I enjoy reading, performing magic tricks and playing the piano, guitar, and occasionally the drums.

Kief Morris

I am the Continuous Delivery Practice Lead for ThoughtWorks Europe..

I specialize in tools, practices, and processes for the continuous delivery of software.

Sudhindra Rao

I've been a Ruby developer for the last 6 years. I've developed apps for a variety of domains - publishing, retail and auctions, large scale datacenter management, and even one to find voters for campaigning during elections. I was part of the team that built the SaaS offering for Mingle.

I am passionate about contributing to opensource projects.. I recently worked on building mobile applications and training the future generation of ThoughtWorkers to be great developers and consultants.

ThoughtWorks®

Love the ebook?

For interesting challenging thoughts and insights from the trenches of software development...

Read on

Share this ebook.