Managing Your AWS Infrastructure at Scale

Shaun Pearce

Steven Bryen

February 2015

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

Amazon Web Services (AWS) enables organizations to deploy large-scale application infrastructures across multiple geographic locations. When deploying these large, cloud-based applications, it's important to ensure that the cost and complexity of operating such systems does not increase in direct proportion to their size.

This whitepaper is intended for existing and potential customers—especially architects, developers, and sysops administrators—who want to deploy and manage their infrastructure in a scalable and predictable way on AWS.

In this whitepaper, we describe tools and techniques to provision new instances, configure the instances to meet your requirements, and deploy your application code. We also introduce strategies to ensure that your instances remain stateless, resulting in an architecture that is more scalable and fault tolerant. The techniques we describe allow you to scale your service from a single instance to thousands of instances while maintaining a consistent set of processes and tools to manage them.

For the purposes of this whitepaper, we assume that you have knowledge of basic scripting and core services such as Amazon Elastic Compute Cloud (Amazon EC2).

Introduction

When designing and implementing large, cloud-based applications, it's important to consider how your infrastructure will be managed to ensure the cost and complexity of running such systems is minimized. When you first begin using Amazon EC2, it is easy to manage your EC2 instances just like regular virtualized servers running in your data center. You can create an instance, log in, configure the operating system, install any additional packages, and install your application code. You can maintain the instance by installing security patches, rolling out new deployments of your code, and modifying the configuration as needed. Despite the operational overhead, you can continue to manage your instances in this way for a long time.

However, your instances will inevitably begin to diverge from their original specification, which can lead to inconsistencies with other instances in the same environment. This divergence from a known baseline can become a huge challenge when managing large fleets of instances across multiple environments. Ultimately, it will lead to service issues because your environments will become less predictable and more difficult to maintain.

The AWS platform provides you with a set of tools to address this challenge with a different approach. By using Amazon EC2 and associated services, you can specify and manage the desired end state of your infrastructure independently of the EC2 instances and other running components.

For example, with a traditional approach you would alter the configuration of an Apache server running across your web servers by logging in to each server in turn and manually making the change. By using the AWS platform, you can take a different approach by changing the underlying specification of your web servers and launching new EC2 instances to replace the old ones. This ensures that each instance remains identical; it also reduces the effort to implement the change and reduces the likelihood of errors being introduced.

When you start to think of your infrastructure as being defined independently of the running EC2 instances and other components in your environments, you can take greater advantage of the benefits of dynamic cloud environments:

- **Software-defined infrastructure** – By defining your infrastructure using a set of software artifacts, you can leverage many of the tools and techniques that are used when developing software components. This includes managing the evolution of your infrastructure in a version control system, as well as using continuous integration (CI) processes to continually test and validate infrastructure changes before deploying them to production.
- **Auto Scaling and self-healing** – If you automatically provision your new instances from a consistent specification, you can use Auto Scaling groups to manage the number of instances in an EC2 fleet. For example, you can set a condition to add new EC2 instances in increments to the Auto Scaling group when the average utilization of your EC2 fleet is high. You can also use Auto Scaling to detect impaired EC2 instances and unhealthy applications, and replace the instances without your intervention.
- **Fast environment provisioning** – You can quickly and easily provision consistent environments, which opens up new ways of working within your teams. For example, you can provision a new environment to allow testers to validate a new version of your application in their own, personal test environments that are isolated from other changes.
- **Reduce costs** – Now that you can provision environments quickly, you also have the option to remove them when they are no longer needed. This reduces costs because you pay only for the resources that you use.
- **Blue-green deployments** – You can deploy new versions of your application by provisioning new instances (containing a new version of the code) beside your existing infrastructure. You can then switch traffic between environments in an approach known as blue-green deployments. This has many benefits over traditional deployment strategies, including the ability to quickly and easily roll back a deployment in the event of an issue.

To leverage these advantages, your infrastructure must have the following capabilities:

1. New infrastructure components are automatically provisioned from a known, version-controlled baseline in a repeatable and predictable manner
2. All instances are stateless so that they can be removed and destroyed at any time, without the risk of losing application state or system data

The following figure shows the overall process:

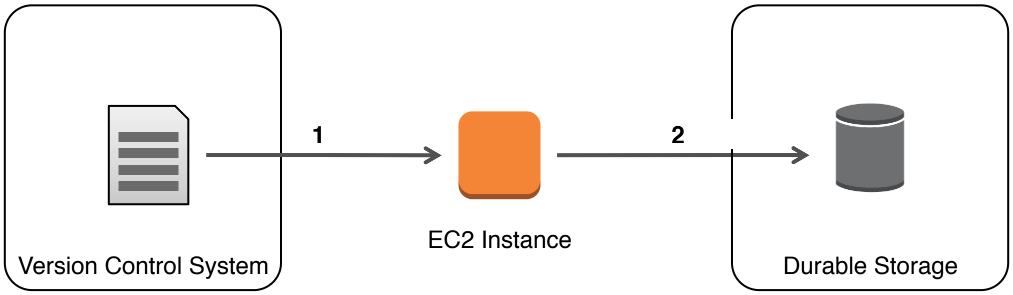

Figure 1: Instance Lifecycle and State Management

The following sections outline tools and techniques that you can use to build a system with these capabilities. By moving to an architecture where your instances can be easily provisioned and destroyed with no loss of data, you can fundamentally change the way you manage your infrastructure. Ultimately, you can scale your infrastructure over time without significantly increasing the operational overhead associated with it.

Provisioning New EC2 Instances

A number of external events will require you to provision new instances into your environments:

- Creating new instances or replicating existing environments
- Replacing a failed instance in an existing environment
- Responding to a “scale up” event to add additional instances to an Auto Scaling group
- Deploying a new version of your software stack (by using blue-green deployments)

Some of these events are difficult or even impossible to predict, so it’s important that the process to create new instances into your environment is fully automated, repeatable, and consistent.

The process of automatically provisioning new instances and bringing them into service is known as *bootstrapping*. There are multiple approaches to bootstrapping your Amazon EC2 instances. The two most popular approaches are to either create your own

Amazon Machine Image (AMI) or to use dynamic configuration. We explain both approaches in the following sections.

Creating Your Own AMI

An Amazon Machine Image (AMI) is a template that provides all of the information required to launch an Amazon EC2 instance. At a minimum it contains the base operating system, but it may also include additional configuration and software. You can launch multiple instances of an AMI, and you can also launch different types of instances from a single AMI.

You have several options when launching a new EC2 instance:

- Select an AMI provided by AWS
- Select an AMI provided by the community
- Select an AMI containing pre-configured software from the AWS Marketplace[1]
- Create a custom AMI

If launching an instance from a base AMI containing only the operating system, you can further customize the instance with additional configuration and software after it has been launched. If you create a custom AMI, you can launch an instance that already contains your complete software stack, thereby removing the need for any run-time configuration. However, before you decide whether to create a custom AMI, you should understand the advantages and disadvantages.

Advantages of custom AMIs

- **Increases speed –** All configuration is packaged into the AMI itself, which significantly increases the speed in which new instances can be launched. This is particularly useful during Auto Scaling events.
- **Reduces external dependencies –** Packaging everything into an AMI means that there is no dependency on the availability of external services when launching new instances (for example, package or code repositories).
- **Removes the reliance on complex configuration scripts at launch time –** By preconfiguring your AMI, scaling events and instance replacements no longer rely on the successful completion of configuration scripts at launch time. This reduces the likelihood of operational issues caused by erroneous scripts.

Disadvantages of custom AMIs

[1] https://aws.amazon.com/marketplace

- **Loss of agility** – Packaging everything into an AMI means that even simple code changes and defect fixes will require you to produce a new AMI. This increases the time it takes to develop, test, and release enhancements and fixes to your application.
- **Complexity –** Managing the AMI build process can be complex. You need a process that enables the creation of consistent, repeatable AMIs where the changes between revisions are identifiable and auditable.
- **Run-time configuration requirements –** You might need to make additional customizations to your AMIs based on run-time information that cannot be known at the time the AMI is created. For example, the database connection string required by your application might change depending on where the AMI is used.

Given these advantages and disadvantages, we recommend a hybrid approach: build static components of your stack into AMIs, and configure dynamic aspects that change regularly (such as application code) at run time.

Consider the following factors to help you decide what configuration to include within a custom AMI and what to include in dynamic run-time scripts:

- **Frequency of deployments** – How often are you likely to deploy enhancements to your system, and at what level in your stack will you make the deployments? For example, you might deploy changes to your application on a daily basis, but you might upgrade your JVM version far less frequently.
- **Reduction on external dependencies** – If the configuration of your system depends on other external systems, you might decide to carry out these configuration steps as part of an AMI build rather than at the time of launching an instance.
- **Requirements to scale quickly –** Will your application use Auto Scaling groups to adjust to changes in load? If so, how quickly will the load on the application increase? This will dictate the speed in which you need to provision new instances into your EC2 fleet.

Once you have assessed your application stack based on the preceding criteria, you can decide which elements of your stack to include in a custom AMI and which will be configured dynamically at the time of launch.

The following figure shows a typical Java web application stack and how it could be managed across AMIs and dynamic scripts.

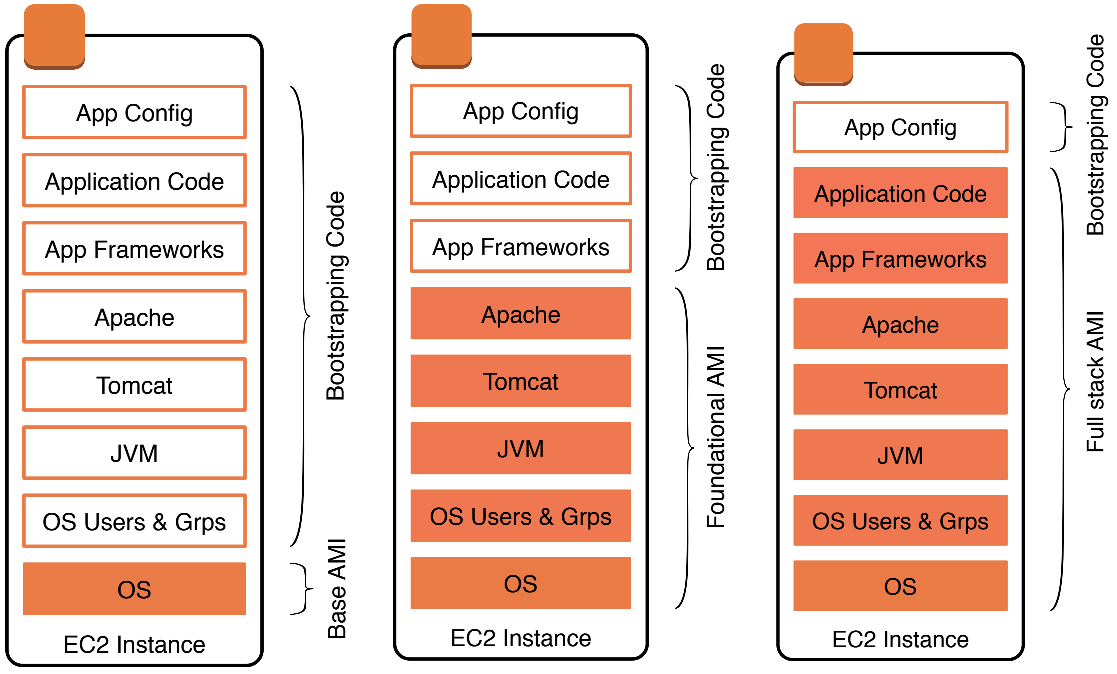

Figure 2: Base, Foundational, and Full AMI Models

In the base AMI model, only the OS image is maintained as an AMI. The AMI can be an AWS-managed image, or an AMI that you manage that contains your own OS image.

In the foundational AMI model, elements of a stack that change infrequently (for example, components such as the JVM and application server) are built into the AMI.

In the full stack AMI model, all elements of the stack are built into the AMI. This model is useful if your application changes infrequently, or if your application has rapid auto-scaling requirements (which means that dynamically installing the application isn't feasible). However, even if you build your application into the AMI, it still might be advantageous to dynamically configure the application at run time because it increases the flexibility of the AMI. For example, it enables you to use your AMIs across multiple environments.

Managing AMI Builds

Many people start by manually configuring their AMIs using a process similar to the following:

1. Launch the latest version of the AMI
2. Log in to the instance and manually reconfigure it (for example, by making package updates or installing new applications)
3. Create a new AMI based on the running instance

Although this manual process is sufficient for simple applications, it is difficult to manage in more complex environments where AMI updates are needed regularly. It’s essential to have a consistent, repeatable process to create your AMIs. It’s also important to be able to audit what has changed between one version of your AMI and another.

One way to achieve this is to manage the customization of a base AMI by using automated scripts. You can develop your own scripts, or you can use a configuration management tool. For more information about configuration management tools, see the Using Configuration Management Tools section in this whitepaper.

Using automated scripts has a number of advantages over the manual method. Automation significantly speeds up the AMI creation process. In addition, you can use version control for your scripts/configuration files, which results in a repeatable process where the change between AMI versions is transparent and auditable.

This automated process is similar to the manual process:

1. Launch the latest version of the AMI
2. Execute the automated configuration using your tool of choice
3. Create a new AMI image based on the running instance

You can use a third-party tool such as Packer[2] to help automate the process. However, many find that this approach is still too time consuming for an environment with multiple, frequent AMI builds across multiple environments.

If you use the Linux operating system, you can reduce the time it takes to create a new AMI by customizing an Amazon Elastic Block Store (Amazon EBS) volume rather than a running instance. An Amazon EBS volume is a durable, block-level storage device that you can attach to a single Amazon EC2 instance. It is possible to create an Amazon EBS volume from a base AMI snapshot and customise this volume before storing it as a new AMI. This replaces the time taken to initialize an EC2 instance with the far shorter time needed to create and attach an EBS volume.

In addition, this approach makes use of the incremental nature of Amazon EBS snapshots. An EBS snapshot is a point-in-time backup of an EBS volume that is stored in Amazon S3. Snapshots are incremental backups, meaning that only the blocks on the device that have changed after your most recent snapshot are saved. For example, if a configuration update changes only 100 MB of the blocks on an 8 GB EBS volume, only 100 MB will be stored to Amazon S3.

[2] https://packer.io

To achieve this, you need a long running EC2 instance that is responsible for attaching a new EBS volume based on the latest AMI build, executing the scripts needed to customize the volume, creating a snapshot of the volume, and registering the snapshot as a new version of your AMI. For example, Netflix uses this technique in their open source tool called aminator.[3]

The following figure shows this process.

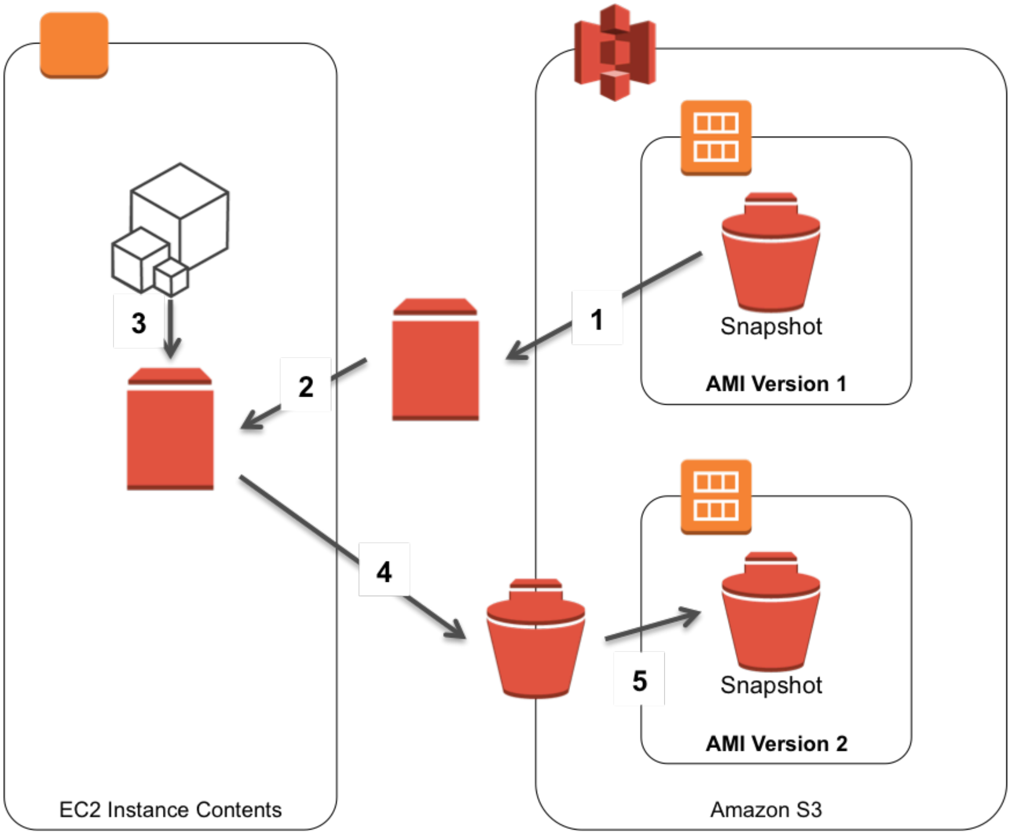

Figure 3: Using EBS Snapshots to Speed Up Deployments

1. Create the volume from the latest AMI snapshot
2. Attach the volume to the instance responsible for building new AMIs
3. Run automated provisioning scripts to update the AMI configuration
4. Snapshot the volume
5. Register the snapshot as a new version of the AMI

[3] https://github.com/Netflix/aminator

Dynamic Configuration

Now that you have decided what to include into your AMI and what should be dynamically configured at run time, you need to decide how to complete the dynamic configuration and bootstrapping process. There are many tools and techniques that you can use to configure your instances, ranging from simple scripts to complex, centralized configuration management tools.

Scripting Your Own Solution

Depending on how much pre-configuration has been included into your AMI, you might need only a single script or set of scripts as a simple, elegant way to configure the final elements of your application stack.

User Data and cloud-init

When you launch a new EC2 instance by using either the AWS Management Console or the API, you have the option of passing user data to the instance. You can retrieve the user data from the instance through the EC2 metadata service, and use it to perform automated tasks to configure instances as they are first launched.

When a Linux instance is launched, the initialization instructions passed into the instance by means of the user data are executed by using a technology called `cloud-init`. The `cloud-init` package is an open source application built by Canonical. It's included in many base Linux AMIs (to find out if your distribution supports `cloud-init`, see the distribution-specific documentation). Amazon Linux, a Linux distribution created and maintained by AWS, contains a customized version of `cloud-init`.

You can pass two types of user data, either shell scripts or `cloud-init` directives, to `cloud-init` running on your EC2 instance. For example, the following shell script can be passed to an instance to update all installed packages and to configure the instance as a PHP web server:

```
#!/bin/sh
yum update -y
yum -y install httpd php php-mysql
chkconfig httpd on
/etc/init.d/httpd start
```

The following user data achieves the same result, but uses a set of `cloud-init` directives:

```
#cloud-config
```

```
repo_update: true
repo_upgrade: all
```

```
packages:
 - httpd
 - php
 - php-mysql

runcmd:
 - service httpd start
 - chkconfig httpd on
```

AWS Windows AMIs contain an additional service, EC2Config, that is installed by AWS. The EC2Config service performs tasks on the instance such as activating Windows, setting the Administrator password, writing to the AWS console, and performing one-click sysprep from within the application. If launching a Windows instance, the EC2Config service can also execute scripts passed to the instance by means of the user data. The data can be in the form of commands that you run at the cmd.exe prompt or Windows PowerShell prompt.

This approach works well for simple use cases. However, as the number of instance roles (web, database, and so on) grows along with the number of environments that you need to manage, your scripts might become large and difficult to maintain. Additionally, user data is limited to 16 KB, so if you have a large number of configuration tasks and associated logic, we recommend that you use the user data to download additional scripts from Amazon S3 that can then be executed.

Leveraging EC2 Metadata

When you configure a new instance, you typically need to understand the context in which the instance is being launched. For example, you might need to know the hostname of the instance or which region or Availability Zone the instance has been launched into. The EC2 metadata service can be queried to provide such contextual information about an instance, as well as retrieving the user data. To access the instance metadata from within a running instance, you can make a standard HTTP GET using tools such as cURL or the GET command. For example, to retrieve the host name of the instance, you can make an HTTP GET request to the following URL:

```
http://169.254.169.254/latest/meta-data/hostname
```

Resource Tagging

To help you manage your EC2 resources, you can assign your own metadata to each instance in addition to the EC2 metadata that is used to define hostnames, Availability Zones, and other resources. You do this with *tags*. Each tag consists of a key and a value, both of which you define when the instance is launched.

You can use EC2 tags to define further context to the instance being launched. For example, you can tag your instances for different environments and roles, as shown in the following figure.

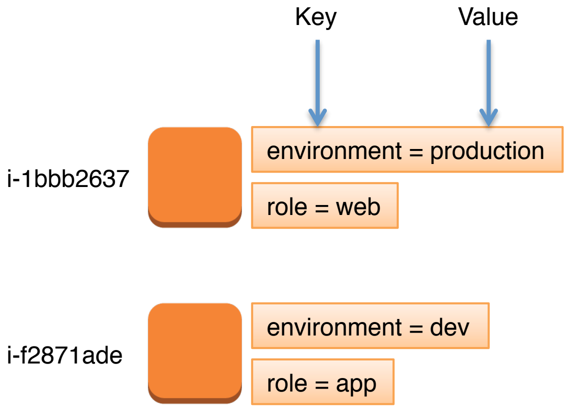

Figure 4: Example of EC2 Tag Usage

As long as your EC2 instance has access to the Internet, these tags can be retrieved by using the AWS Command Line Interface (CLI) within your bootstrapping scripts to configure your instances based on their role and the environment in which they are being launched.

Putting it all Together

The following figure shows a typical bootstrapping process using user data and a set of configuration scripts hosted on Amazon S3.

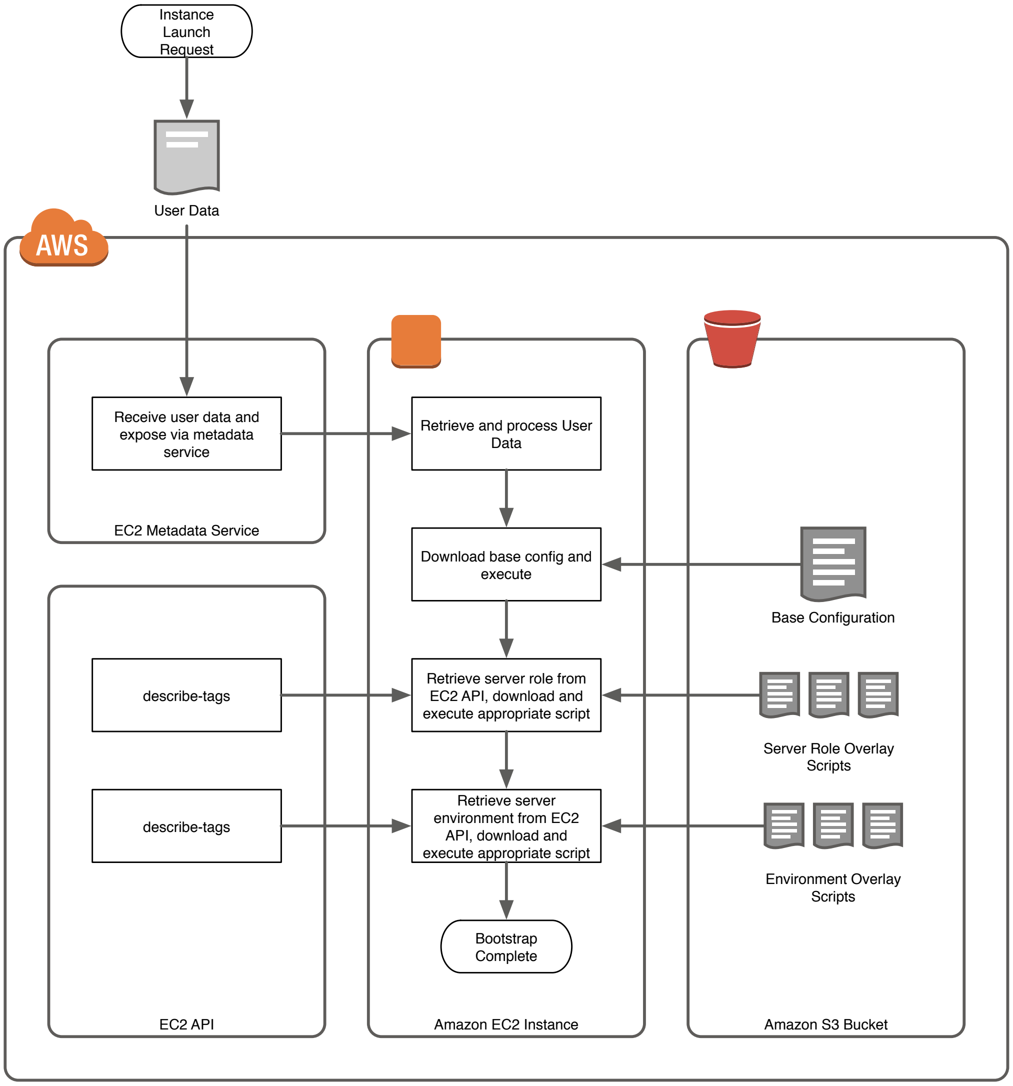
Instance Launch Request
User Data
AWS
Receive user data and expose via metadata service
EC2 Metadata Service
Retrieve and process User Data
Download base config and execute
Base Configuration
describe-tags
Retrieve server role from EC2 API, download and execute appropriate script
Server Role Overlay Scripts
describe-tags
Retrieve server environment from EC2 API, download and execute appropriate script
Environment Overlay Scripts
Bootstrap Complete
EC2 API
Amazon EC2 Instance
Amazon S3 Bucket

Figure 5: Example of an End-to-End Workflow

This example uses the user data as a lightweight mechanism to download a base configuration script from Amazon S3. The script is responsible for configuring the system to a baseline across all instances regardless of role and environment (for example, the script might install monitoring agents and ensure that the OS is patched).

This base configuration script uses the CLI to retrieve the instances tags. Based on the value of the "role" tag, the script downloads an additional overlay script responsible for the additional configuration required for the instance to perform its specific role (for example, installing Apache on a web server). Finally, the script uses the instances "environment" tag to download an appropriate environment overlay script to carry out the final configuration for the environment the instance resides in (for example, setting log levels to DEBUG in the development environment).

To protect sensitive information that might be contained in your scripts, you should restrict access to these assets by using IAM Roles.[4]

Using Configuration Management Tools

Although scripting your own solution works, it can quickly become complex when managing large environments. It also can become difficult to govern and audit your environment, such as identifying changes or troubleshooting configuration issues. You can address some of these issues by using a configuration management tool to manage instance configurations.

Configuration management tools allow you to define your environment's configuration in code, typically by using a domain-specific language. These domain-specific languages use a declarative approach to code, where the code describes the end state and is not a script that can be executed. Because the environment is defined using code, you can track changes to the configuration and apply version control. Many configuration management tools also offer additional features such as compliance, auditing, and search.

Push vs. Pull Models

Configuration management tools typically leverage one of two models, push or pull. The model used by a tool is defined by how a node (a target EC2 instance in AWS) interacts with the master configuration management server.

In a push model, a master configuration management server is aware of the nodes that it needs to manage and pushes the configuration to them remotely. These nodes need to be pre-registered on the master server. Some push tools are agentless and execute configuration remotely using existing protocols such as SSH. Others push a package, which is then executed locally using an agent. The push model typically has some constraints when working with dynamic and scalable AWS resources:

- The master server needs to have information about the nodes that it needs to manage. When you use tools such as Auto Scaling, where nodes might come and go, this can be a challenge.
- Push systems that do remote execution do not scale as well as systems where configuration changes are offloaded and executed locally on a node. In large

[4] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html

environments, the master server might get overloaded when configuring multiple systems in parallel.

- Connecting to nodes remotely requires you to allow specific ports to be allowed inbound to your nodes. For some remote execution tools, this includes remote SSH.

The second model is the pull model. Configuration management tools that use a pull system use an agent that is installed on a node. The agent asks the master server for configuration. A node can pull its configuration at boot time, or agents can be daemonized to poll the master periodically for configuration changes. Pull systems are especially useful for managing dynamic and scalable AWS environments. Following are the main benefits of the pull model:

- Nodes can scale up and down easily, as the master does not need to know they exist before they can be configured. Nodes can simply register themselves with the server.
- Configuration management masters require less scaling when using a pull system because all processing is offloaded and executed locally on the remote node.
- No specific ports need to be opened inbound to the nodes. Most tools allow the agent to communicate with the master server by using typical outbound ports such as HTTPS.

Chef Example

Many configuration management tools work with AWS. Some of the most popular are Chef, Puppet, Ansible, and SaltStack. For our example in this section, we use Chef to demonstrate bootstrapping with a configuration management tool. You can use other tools and apply the same principles.

Chef is an open source configuration management tool that uses an agent (chef-client) to pull configuration from a master server (Chef server). Our example shows how to bootstrap nodes by pulling configuration from a Chef server at boot time.

The example is based on the following assumptions:

- You have configured a Chef server
- You have an AMI that has the AWS command line tools installed and configured
- You have the chef-client installed and included into your AMI

First, let’s look at what we are going to configure within Chef. We’ll create a simple Chef cookbook that installs an Apache web server and deploys a ‘Hello World’ site. A Chef cookbook is a collection of recipes; a recipe is a definition of resources that should be configured on a node. This can include files, packages, permissions, and more. The default recipe for this Apache cookbook might look something like this:

```
#
# Cookbook Name:: apache
# Recipe:: default
#
# Copyright 2014, YOUR_COMPANY_NAME
#
# All rights reserved - Do Not Redistribute
#
package "httpd"

#Allow Apache to start on boot
service "httpd" do
  action [:enable, :start]
end

#Add HTML Template into Web Root
template "/var/www/html/index.html" do
  source "index.html.erb"
  mode "0644"
end
```

In this recipe, we install, enable, and start the HTTPD (HTTP daemon) service. Next, we render a template for index.html and place it into the /var/www/html directory. The index.html.erb template in this case is a very simple HTML page:

```
<h1>Hello World</h1>
```

Next, the cookbook is uploaded to the Chef server. Chef offers the ability to group cookbooks into roles. Roles are useful in large-scale environments where servers within your environment might have many different roles, and cookbooks might have overlapping roles. In our example, we add this cookbook to a role called 'webserver'.

Now when we launch EC2 instances (nodes), we can provide EC2 user data to bootstrap them by using Chef. To make this as dynamic as possible, we can use an EC2 tag to define which Chef role to apply to our node. This allows us to use the same user data script for all nodes, whichever role is intended for them. For example, a web server and a database server can use the same user data if you assign different values to the 'role' tag in EC2.

We also need to consider how our new instance will authenticate with the Chef server. We can store our private key in an encrypted Amazon S3 bucket by using Amazon S3 server side encryption,[5] and we can restrict access to this bucket by using IAM roles. The key can then be used to authenticate with the Chef server. The chef-client uses a validator.pem file to authenticate to the Chef server when registering new nodes.

We also need to know which Chef server to pull our configuration from. We can store a pre-populated client.rb file in Amazon S3 and copy this within our user data script. You might want to dynamically populate this client.rb file depending on environment, but for our example we assume that we have only one Chef server and that a pre-populated client.rb file is sufficient. You could also include these two files into your custom AMI build.

The user data would look like this:

```
#!/bin/bash
cd /etc/chef

#Copy Chef Server Private Key from S3 Bucket
aws s3 cp s3://s3-bucket/orgname-validator.pem orgname-
validator.pem

#Copy Chef Client Configuration File from S3 Bucket
aws s3 cp s3://s3-bucket/client.rb client.rb

#Change permissions on Chef Server private key.
chmod 400 /etc/chef/orgname-validator.pem

#Get EC2 Instance ID from the Meta-Data Service
INSTANCE_ID=`curl -s http://169.254.169.254/latest/meta-
data/instance-id`

#Get Tag with Key of ‘role’ for this EC2 instance
ROLE_TAG=$(aws ec2 describe-tags --filters "Name=resource-
id,Values=$INSTANCE_ID" "Name=key,Values=role" --output
text)

#Get value of Tag with Key of ‘role’ as string
ROLE_TAG_VALUE=$(echo $ROLE_TAG | awk 'NF>1{print $NF}')

#Create first_boot.json file dynamically adding the tag
value as the chef role in the run-list
echo "{\"run_list\":[\"role[$ROLE_TAG_VALUE]\"]}" >
first_boot.json
```

[5] http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingServerSideEncryption.html

```
#execute the chef-client using first_boot.json config
chef-client -j first_boot.json
```

```
#daemonize the chef-client to run every 5 minutes
chef-client -d -i 300 -s 30
```

As shown in the preceding user data example, we copy our client configuration files from a private S3 bucket. We then use the EC2 metadata service to get some information about the instance (in this example, Instance ID). Next, we query the Amazon EC2 API for any tags with the key of 'role,' and dynamically configure a Chef run-list with a Chef role of this value. Finally, we execute the first chef-client run by providing the first_boot.json options, which include our new run-list. We then execute chef-client once more; however, this time we execute it in a daemonized setup to pull configuration every 5 minutes.

We now have some re-usable EC2 user data that we can apply to any new EC2 instances. As long as a 'role' tag is provided with a value that matches a role on the target Chef server, the instance will be configured using the corresponding Chef cookbooks.

Putting it all Together

The following figure shows a typical workflow, from instance launch to a fully configured instance that is ready to serve traffic.

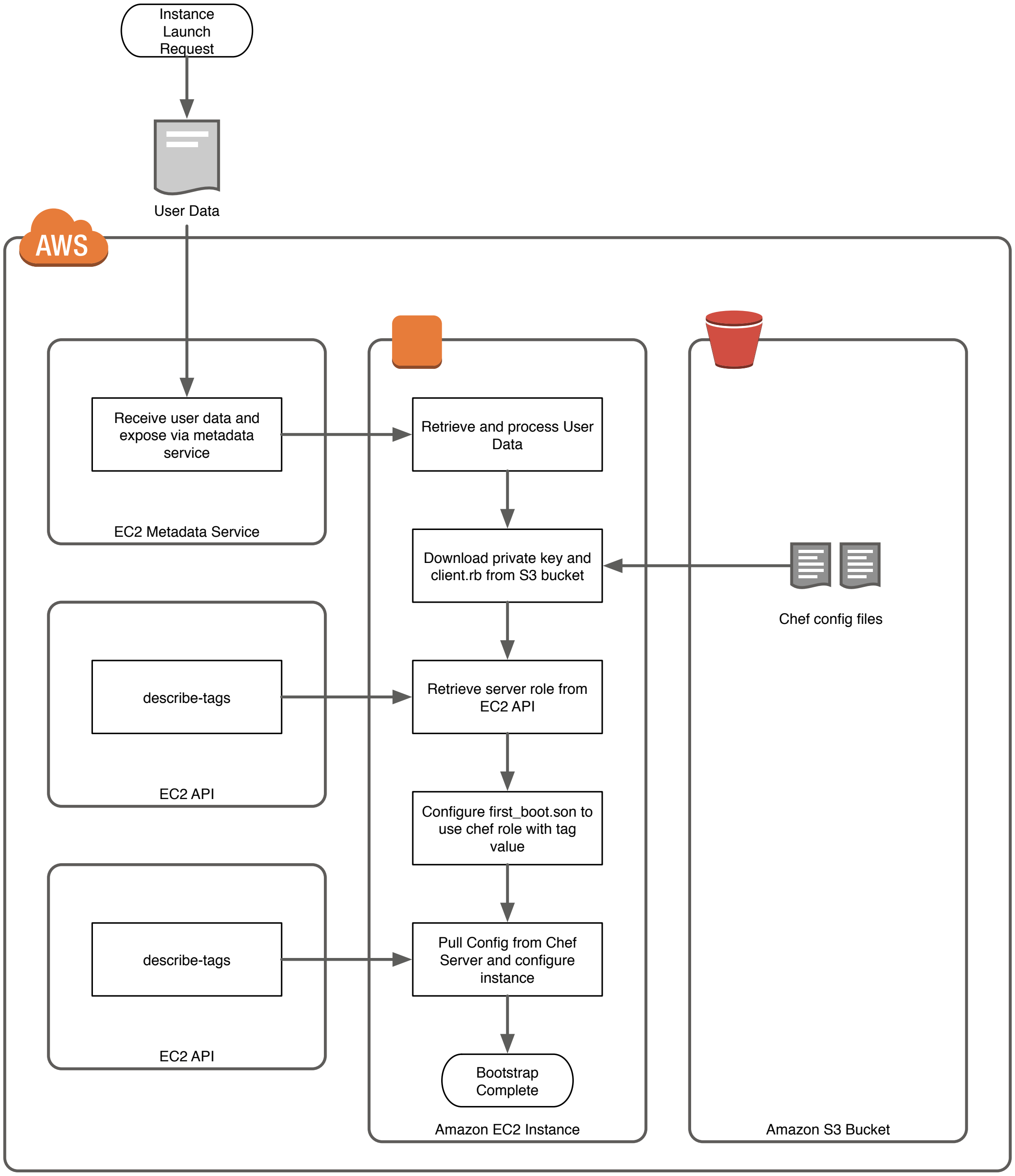
Instance Launch Request
User Data
AWS
Receive user data and expose via metadata service
EC2 Metadata Service
Retrieve and process User Data
Download private key and client.rb from S3 bucket
Chef config files
describe-tags
EC2 API
Retrieve server role from EC2 API
Configure first_boot.son to use chef role with tag value
describe-tags
EC2 API
Pull Config from Chef Server and configure instance
Bootstrap Complete
Amazon EC2 Instance
Amazon S3 Bucket

Figure 6: Example of an End-to-End Workflow

Using AWS Services to Help Manage Your Environments

In the preceding sections, we discussed tools and techniques that systems administrators and developers can use to provision EC2 instances in an automated, predictable, and repeatable manner. AWS also provides a range of application management services that help make this process simpler and more productive. The following figure shows how to select the right service for your application based on the level of control that you require.

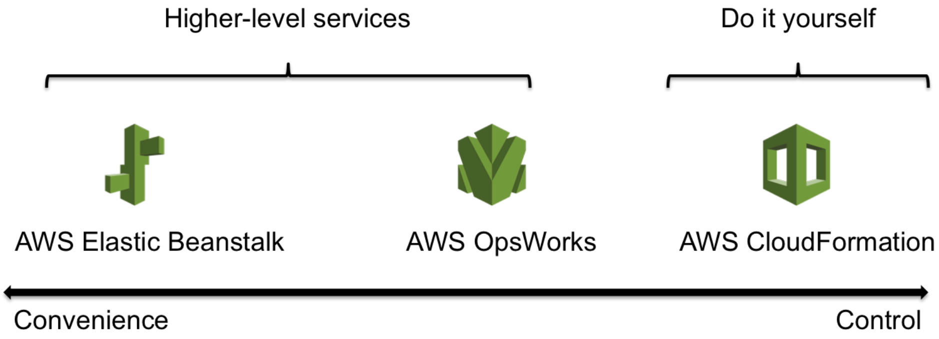

Figure 7: AWS Deployment and Management Services

In addition to provisioning EC2 instances, these services can also help you to provision any other associated AWS components that you need in your systems, such as Auto Scaling groups, load balancers, and networking components. We provide more information about how to use these services in the following sections.

AWS Elastic Beanstalk

AWS Elastic Beanstalk allows web developers to easily upload code without worrying about managing or implementing any underlying infrastructure components. Elastic Beanstalk takes care of deployment, capacity provisioning, load balancing, auto scaling, and application health monitoring. It is worth noting that Elastic Beanstalk is not a black box service: You have full visibility and control of the underlying AWS resources that are deployed, such as EC2 instances and load balancers.

Elastic Beanstalk supports deployment of Java, .NET, Ruby, PHP, Python, Node.js, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS. Elastic Beanstalk provides a default configuration, but you can extend the configuration as needed. For example, you might want to install additional packages from a **yum** repository or copy configuration files that your application depends on, such as a replacement for httpd.conf to override specific settings.

You can write the configuration files in YAML or JSON format and create the files with a .config file extension. You then place the files in a folder in the application root named .ebextensions. You can use configuration files to manage packages and services, work with files, and execute commands.

For more information about using and extending Elastic Beanstalk, see AWS Elastic Beanstalk Documentation.[6]

AWS OpsWorks

AWS OpsWorks is an application management service that makes it easy to deploy and manage any application and its required AWS resources. With AWS OpsWorks, you build application stacks that consist of one or many layers. You configure a layer by using an AWS OpsWorks configuration, a custom configuration, or a mix of both. AWS OpsWorks uses Chef, the open source configuration management tool, to configure AWS resources. This gives you the ability to provide your own custom or community Chef recipes.

AWS OpsWorks features a set of lifecycle events—Setup, Configure, Deploy, Undeploy, and Shutdown—that automatically run the appropriate recipes at the appropriate time on each instance. AWS OpsWorks provides some AWS-managed layers for typical application stacks. These layers are open and customizable, which means that you can add additional custom recipes to the layers provided by AWS OpsWorks or create custom layers from scratch using your existing recipes.

It is important to ensure that the correct recipes are associated with the correct lifecycle events. Lifecycle events run during the following times:

- Setup – Occurs on a new instance after it successfully boots
- Configure – Occurs on all of the stack's instances when an instance enters or leaves the online state
- Deploy – Occurs when you deploy an app
- Undeploy – Occurs when you delete an app
- Shutdown – Occurs when you stop an instance

For example, the configure event is useful when building distributed systems or for any system that needs to be aware of when new instances are added or removed from the stack. You could use this event to update a load balancer when new web servers are added to the stack.

[6] http://aws.amazon.com/documentation/elastic-beanstalk/

In addition to typical server configuration, AWS OpsWorks manages application deployment and integrates with your application's code repository. This allows you to track application versions and rollback deployments if needed.

For more information about AWS OpsWorks, see AWS OpsWorks Documentation.[7]

AWS CloudFormation

AWS CloudFormation gives developers and systems administrators an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion. Compared to Elastic Beanstalk and AWS OpsWorks, AWS CloudFormation gives you the most control and flexibility when provisioning resources.

AWS CloudFormation allows you to manage a broad set of AWS resources. For the purposes of this whitepaper, we focus on the features that you can use to bootstrap your EC2 instances.

User Data

Earlier in this whitepaper, we described the process of using user data to configure and customize your EC2 instances (see Scripting Your Own Solution). You also can include user data in an AWS CloudFormation template, which is executed on the instance once it is created. You can include user data when specifying a single EC2 instance as well as when specifying a launch configuration. The following example shows a launch configuration that provisions instances configured to be PHP web servers:

```
"MyLaunchConfig" : {
  "Type" : "AWS::AutoScaling::LaunchConfiguration",
  "Properties" : {
    "ImageId" : "i-123456",
    "SecurityGroups" : "MySecurityGroup",
    "InstanceType" : "m3.medium",
    "KeyName" : "MyKey",
    "UserData": {"Fn::Base64": {"Fn::Join":["",[
      "#!/bin/bash\n",
      "yum update -y\n",
      "yum -y install httpd php php-mysql\n",
      "chkconfig httpd on\n",
      "/etc/init.d/httpd start\n"
    ]]}}
```

[7] http://aws.amazon.com/documentation/opsworks/

```
  }
}
```

cfn-init

The cfn-init script is an AWS CloudFormation helper script that you can use to specify the end state of an EC2 instance in a more declarative manner. The cfn-init script is installed by default on Amazon Linux and AWS-supplied Windows AMIs. Administrators can also install cfn-init on other Linux distributions, and then include this into their own AMI if needed.

The cfn-init script parses metadata from the AWS CloudFormation template and uses the metadata to customize the instance accordingly. The cfn-init script can do the following:

- Install packages from package repositories (such as **yum** and **apt-get**)
- Download and unpack archives, such as .zip and .tar files
- Write files to disk
- Execute arbitrary commands
- Create users and groups
- Enable/disable and start/stop services

In an AWS CloudFormation template, the cfn-init helper script is called from the user data. Once it is called, it will inspect the metadata associated with the resource passed into the request and then act accordingly. For example, you can use the following launch configuration metadata to instruct cfn-init to configure an EC2 instance to become a PHP web server (similar to the preceding user data example):

```
"MyLaunchConfig" : {
  "Type" : "AWS::AutoScaling::LaunchConfiguration",
  "Metadata" : {
    "AWS::CloudFormation::Init" : {
      "config" : {
        "packages" : {
          "yum" : {
            "httpd" : [],
            "php" : [],
            "php-mysql" : []
          }
        },
        "services" : {
          "sysvinit" : {
            "httpd" : {
              "enabled"       : "true",
              "ensureRunning" : "true"
```

```
            }
          }
        }
      }
    }
  },
  "Properties" : {
    "ImageId" : "i-123456",
    "SecurityGroups" : "MySecurityGroup",
    "InstanceType" : "m3.medium",
    "KeyName" : "MyKey",
    "UserData": {"Fn::Base64": {"Fn::Join":["",[
      "#!/bin/bash\n",
      "yum update -y aws-cfn-bootstrap\n",
      "/opt/aws/bin/cfn-init --stack ", { "Ref" :
"AWS::StackId" }, " --resource MyLaunchConfig ",
      "    --region ", { "Ref" : "AWS::Region" }, "\n",
    ]]}}
  }
}
```

For a detailed walkthrough of bootstrapping EC2 instances by using AWS CloudFormation and its related helper scripts, see the Bootstrapping Applications via AWS CloudFormation whitepaper.[8]

Using the Services Together

You can use the services separately to help you provision new infrastructure components, but you also can combine them to create a single solution. This approach has clear advantages. For example, you can model an entire architecture, including networking and database configurations, directly into an AWS CloudFormation template, and then deploy and manage your application by using AWS Elastic Beanstalk or AWS OpsWorks. This approach unifies resource and application management, making it easier to apply version control to your entire architecture.

[8] https://s3.amazonaws.com/cloudformation-examples/BoostrappingApplicationsWithAWSCloudFormation.pdf

Managing Application and Instance State

After you implement a suitable process to automatically provision new infrastructure components, your system will have the capability to create new EC2 instances and even entire new environments in a quick, repeatable, and predictable manner. However, in a dynamic cloud environment you will also need to consider how to remove EC2 instances from your environments, and what impact this might have on the service that you provide to your users. There are a number of reasons why an instance might be removed from your system:

- The instance is terminated as a result of a hardware or software failure
- The instance is terminated as a response to a "scale down" event to remove instances from an Auto Scaling group
- The instance is terminated because you've deployed a new version of your software stack by using blue-green deployments (instances running the older version of the application are terminated after the deployment)

To handle the removal of instances without impacting your service, you need to ensure that your application instances are *stateless*. This means that all system and application state is stored and managed outside of the instances themselves. There are many forms of system and application state that you need to consider when designing your system, as shown in the following table.

State	Examples
Structured application data	Customer orders
Unstructured application data	Images and documents
User session data	Position in the app; contents of a shopping cart
Application and system logs	Access logs; security audit logs
Application and system metrics	CPU load; network utilization

Running stateless application instances means that no instance in a fleet is any different from its counterparts. This offers a number of advantages:

- **Providing a robust service –** Instances can serve any request from any user at any time. If an instance fails, subsequent requests can be routed to alternative instances while the failed instance is replaced. This can be achieved with no interruption to service for any of your users.
- **Quicker, less complicated bootstrapping –** Because your instances don't contain any dynamic state, your bootstrapping process needs to concern itself only with provisioning your system up to the application layer. There is no need to try to

recover state and data, which is often large and therefore can significantly increase bootstrapping times.

- **EC2 instances as a unit of deployment** – Because all state is maintained off of the EC2 instances themselves, you can replace the instances while orchestrating application deployments. This can simplify your deployment processes and allow new deployment techniques, such as blue-green deployments.

The following section describes each form of application and instance state, and outlines some of the tools and techniques that you can use to ensure it is stored separately and independently from the application instances themselves.

Structured Application Data

Most applications produce structured, textual data, such as customer orders in an order management system or a list of web pages in a CMS. In most cases, this kind of content is best stored in a database. Depending on the structure of the data and the requirements for access speed and concurrency, you might decide to use a relational database management system or a NoSQL data store. In either case, it is important to store this content in a durable, highly available system away from the instances running your application. This will ensure that the service you provide your users will not be interrupted or their data lost, even in the event of an instance failure.

AWS offers both relational and NoSQL managed databases that you can use as a persistence layer for your applications. We discuss these database options in the following sections.

Amazon RDS

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easy to set up, operate, and scale a relational database in the cloud. It allows you to continue to work with the relational database engines you're familiar with, including MySQL, Oracle, Microsoft SQL Server, or PostgreSQL. This means that the code, applications, and operational tools that you are already using can be used with Amazon RDS. Amazon RDS also handles time-consuming database management tasks, such as data backups, recovery, and patch management, which frees your database administrators to pursue higher value application development or database refinements.

In addition, Amazon RDS Multi-AZ deployments increase your database availability and protect your database against unplanned outages. This gives your service an additional level of resiliency.

Amazon DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service offering both document (JSON) and key-value data models. DynamoDB has been designed to provide consistent, single-digit millisecond latency at any scale, making it ideal for high traffic applications with a requirement for low latency data access. DynamoDB manages the scaling and partitioning of infrastructure on your behalf. When you create a table, you specify how much request capacity you require. If your throughput requirements change, you can update this capacity as needed with no impact on service.

Unstructured Application Data

In addition to the structured data created by most applications, some systems also have a requirement to receive and store unstructured resources such as documents, images, and other binary data. For example, this might be the case in a CMS where an editor uploads images and PDFs to be hosted on a website.

In most cases, a database is not a suitable storage mechanism for this type of content. Instead, you can use Amazon Simple Storage Service (Amazon S3). Amazon S3 provides a highly available and durable object store that is well suited to storing this kind of data. Once your data is stored in Amazon S3, you have the option of serving these files directly from Amazon S3 to your end users over HTTP(S), bypassing the need for these requests to go to your application instances.

User Session Data

Many applications produce information associated with a user's current position within an application. For example, as users browse an e-commerce site, they might start to add various items into their shopping basket. This information is known as *session state*. It would be frustrating to users if the items in their baskets disappeared without notice, so it's important to store the session state away from the application instances themselves. This ensures that baskets remain populated, even if users' requests are directed to an alternative instance behind your load balancer, or if the current instance is removed from service for any reason.

The AWS platform offers a number of services that you can use to provide a highly available session store.

Amazon ElastiCache

Amazon ElastiCache makes it easy to deploy, operate, and scale an in-memory data store in AWS. In-memory data stores are ideal for storing transient session data due to the low latency these technologies offer. ElastiCache supports two open source, in-memory caching engines:

- **Memcached** – A widely adopted memory object caching system. ElastiCache is protocol compliant with Memcached, which is already supported by many open source applications as an in-memory session storage platform.

- **Redis** – A popular open source, in-memory key-value store that supports data structures such as sorted sets and lists. ElastiCache supports master/slave replication and Multi-AZ, which you can use to achieve cross-AZ redundancy.

In addition to the in-memory data stores offered by Memcached and Redis on ElastiCache, some applications require a more durable storage platform for their session data. For these applications, Amazon DynamoDB offers a low latency, highly scalable, and durable solution. DynamoDB replicates data across three facilities in an AWS region to provide fault tolerance in the event of a server failure or Availability Zone outage.

To help customers easily integrate DynamoDB as a session store within their applications, AWS provides pre-built DynamoDB session handlers for both Tomcat-based Java applications[9] and PHP applications.[10]

System Metrics

To properly support a production system, operational teams need access to system metrics that indicate the overall health of the system and the relative load under which it's currently operating. In a traditional environment, this information is often obtained by logging into one of the instances and looking at OS-level metrics such as system load or CPU utilization. However, in an environment where you have multiple instances running, and these instances can appear and disappear at any moment, this approach soon becomes ineffective and difficult to manage. Instead, you should push this data to an external monitoring system for collection and analysis.

Amazon CloudWatch

Amazon CloudWatch is a fully managed monitoring service for AWS resources and the applications that you run on top of them. You can use Amazon CloudWatch to collect and store metrics on a durable platform that is separate and independent from your own infrastructure. This means that the metrics will be available to your operational teams even when the instances themselves have been terminated.

In addition to tracking metrics, you can use Amazon CloudWatch to trigger alarms on the metrics when they pass certain thresholds. You can use the alarms to notify your teams and to initiate further automated actions to deal with issues and bring your system back within its normal operating tolerances. For example, an automated action could initiate an Auto Scaling policy to increase or decrease the number of instances in an Auto Scaling group.

[9] http://docs.aws.amazon.com/AWSSdkDocsJava/latest/DeveloperGuide/java-dg-tomcat-session-manager.html

[10] http://docs.aws.amazon.com/aws-sdk-php/guide/latest/feature-dynamodb-session-handler.html

By default, Amazon CloudWatch can monitor a broad range of metrics across your AWS resources. That said, it is also important to remember that AWS doesn't have access to the OS or applications running on your EC2 instances. Because of this, Amazon CloudWatch cannot automatically monitor metrics that are accessible only within the OS, such as memory and disk volume utilization. If you want to monitor OS and application metrics by using Amazon CloudWatch, you can publish your own metrics to CloudWatch through a simple API request. With this approach, you can manage these metrics in the same way that you manage other, native metrics, including configuring alarms and associated actions.

You can use the EC2Config service[11] to push additional OS-level operating metrics into CloudWatch without the need to manually code against the CloudWatch APIs. If you are running Linux AMIs, you can use the set of sample Perl scripts[12] provided by AWS that demonstrate how to produce and consume Amazon CloudWatch custom metrics.

In addition to CloudWatch, you can use third-party monitoring solutions in AWS to extend your monitoring capabilities.

Log Management

Log data is used by your operational team to better understand how the system is performing and to diagnose any issues that might arise. Log data can be produced by the application itself, but also by system components lower down in your stack. This might include anything from access logs produced by your web server to security audit logs produced by the operating system itself.

Your operations team needs reliable and timely access to these logs at all times, regardless of whether the instance that originally produced the log is still in existence. For this reason, it's important to move log data from the instance to a more durable storage platform as close to real time as possible.

Amazon CloudWatch Logs

Amazon CloudWatch Logs is a service that allows you to quickly and easily move your system and application logs from the EC2 instances themselves to a centralized, durable storage platform (Amazon S3). This ensures that this data is available even when the instance itself has been terminated. You also have control over the log retention policy to ensure that all logs are retained for a specified period of time. The CloudWatch Logs service provides a log management agent that you can install onto your EC2 instances to manage the ingestion of your logs into the log management service.

[11] http://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/UsingConfig_WinAMI.html

[12] http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/mon-scripts-perl.html

In addition to moving your logs to durable storage, the CloudWatch Logs service also allows you to monitor your logs in near real-time for specific phrases, values, or patterns (metrics). You can use these metrics in the same way as any other CloudWatch metrics. For example, you can create a CloudWatch alarm on the number of errors being thrown by your application or when certain, suspect actions are detected in your security audit logs.

Conclusion

This whitepaper showed you how to accomplish the following:

- Quickly provision new infrastructure components in an automated, repeatable, and predictable manner
- Ensure that no EC2 instance in your environment is unique, and that all instances are stateless and therefore easily replaced

Having these capabilities in place allows you to think differently about how you provision and manage infrastructure components when compared to traditional environments. Instead of manually building each instance and maintaining consistency through a set of operational checks and balances, you can treat your infrastructure as if it were software. By specifying the desired end state of your infrastructure through the software-based tools and processes described in this whitepaper, you can fundamentally change the way your infrastructure is managed, and you can take full advantage of the dynamic, elastic, and automated nature of the AWS cloud.

Further Reading

- AWS Elastic Beanstalk Documentation
- AWS OpsWorks Documentation
- Bootstrapping Applications via AWS CloudFormation whitepaper
- Using Chef with AWS CloudFormation
- Integrating AWS CloudFormation with Puppet