# Chapter 7. Deployment Strategies

This chapter covers some methods that can be used to deploy changes safely and reliably to a live environment, and demonstrates two of the common approaches to updating EC2 instances:

Instance-based deployment

Each instance is updated individually, as though it were a server in a traditional datacenter.

AMI-based deployment

A new AMI is created every time a change is released.

The remainder of this chapter investigates the pros and cons of each approach.

In the context of this chapter, deploying does not just refer to updating your application’s code or executable files. It’s a complete, end-to-end process that makes sure your running production environment is consistent and correct. So it covers various other changes you need to manage, such as updating versions of software installed on EC2 instances and making changes to CloudFormation stack templates.

The focus here is on how to orchestrate a fleet of EC2 instances and reliably manage them. I won’t cover the actions taken by the deployment script, such as restarting services because each application has its own unique requirements.

# Instance-Based Deployments

Before we look at the AWS-specific requirements of a deploy system, let’s first examine the components common to nearly all such systems. Consider the deploy flow for a typical software company. We will look at this from the perspective of a developer making a change to some application code, but the same rules apply to a designer changing a CSS file or a sysadmin changing a parameter in a configuration file:

1. A developer writes the code to implement a new feature.
2. The changed files are incorporated into a source control system such as Git or Subversion.
3. Depending on the programming language being used, it might be necessary to build or compile the source files to produce an executable binary.
4. The changed source files (or compiled executables) are made available to the running instances. Instances might pull the files directly from the source control system, or perhaps use a locally hosted repository system such as Apt or Yum.
5. Once the changed files are on the instances, running services are restarted to pick up the new code or configuration files.

This is, of course, a high-level overview, and some applications might require additional steps.

Some parts of the process are not affected by the choice of hosting environment. Whether you are hosting your application on your own hardware in a shared datacenter or on AWS, you will need to perform these steps.

The elasticity of the cloud does enforce some changes onto the standard deploy flow. With Auto Scaling, you can never be sure how many instances will be running when you initiate the deploy process.

Consider this example: you have 10 instances running v1.0 of your application and you wish to deploy an update (v1.1). You run your deploy script and update these 10 instances to v1.1. As word spreads about the amazing new features contained in this release, users flock to the site to join. The increased traffic is noticed by CloudWatch, which responds by launching two new instances to handle the load. Because these instances were not yet launched when the deploy was executed, they will be launched with whichever version of the application was baked into the AMI—in this case, v1.0. As a result, you end up running two versions of your application simultaneously.

To solve this problem, each instance must be able to update itself to the latest released version at launch time, and finish updating itself before it is added to the pool of instances behind an Elastic Load Balancer.

This approach is referred to as an instance-based deploy, and is very similar to release management processes in noncloud environments. Two additional features are required to make it AWS-compatible: finding out the hostnames of instances that should be updated, and making sure instances can update themselves on boot.

## Executing Code on Running Instances with Fabric

The first problem when deploying code to running instances is a simple one: how can we reliably execute code on running EC2 instances when we don’t know their hostnames in advance? Part of the answer to that question is Fabric, which is a Python tool used to automate system administration tasks. It provides a basic set of operations (such as executing commands and transferring files) that can be combined with some custom logic to build powerful and flexible deployment systems, or simply make it easier to perform routine tasks on groups of servers or EC2 instances.

Because Fabric is Python-based, we can use Boto to quickly integrate it with AWS services. Tasks are defined by writing Python functions, which are often stored in a file named fabfile.py. These functions use Fabric’s Python API to perform actions on remote hosts. Here is a simple example:

from fabric.api import run

def get\_uptime():

run('uptime')

When executed, this task will run the uptime command on each host and display the resulting output. It can be executed in various ways, for example:

fab -H localhost,www.example.com get\_uptime

With this invocation, Fabric would execute the get\_uptime task—and therefore theuptime command—on both localhost and www.example.com. The -H flag defines the list of hosts on which the task will be executed.

### GROUPING INSTANCES THROUGH ROLES

Fabric includes a feature known as roles, which are user-defined groups of hosts. The [Roles Documentation](http://docs.fabfile.org/en/1.6/usage/execution.html#roles) shows a simple example:

from fabric.api import env

env.roledefs = {

'web': ['www1', 'www2', 'www3'],

'dns': ['ns1', 'ns2']

}

A role is simply a list of hostnames (or, technically speaking, a list of host strings, that may or may not be fully qualified domain names). As the previous code shows, the role definition list—env.roledefs—is implemented as a Python dictionary, where a key such as 'web' is associated to an array of host strings (www1, www2, and www3). We make it available to other parts of the Python script through the global env variable.

When you combine the code just shown with the previous example, our own get\_uptimetask could be executed on the three web servers by executing this command:

fab -R web get\_uptime

This command would execute get\_uptime on the three web servers listed under env.roledefs as 'web' and display the output, making it functionally equivalent to specifying the three web server hostnames with the -H or --hosts flag.

The previous example relies on a statically defined group of hostnames, which is obviously not suitable for the dynamic nature of EC2. Having to manually create a list of web instances in the production environment before each deployment would quickly become tiresome. Fortunately, role definitions can be set dynamically: before each task is executed, it checks the role definitions to get a list of hostnames. This means one task can update the role definitions, and the updated definitions will be used for all subsequent tasks.

We can use this feature to create different role definitions for staging and production environments. Suppose we are running an infrastructure using hostnames that reference the server’s role and environment. For example, a web server in the production environment is named www1-prod. [Example 7-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch07.html#example_deploy_dynamic_role) shows how dynamic role definitions can be used to control which hosts the tasks are executed on.

##### ***Example 7-1. Dynamic Fabric role definitions***

from fabric.api import env

def production():

env.roledefs = {

'web': ['www1-prod', 'www2-prod', 'www3-prod'],

'db': ['db1-prod', 'db2-prod']

}

def staging():

env.roledefs = {

'web': ['www1-staging', 'www2-staging'],

'db': ['db1-staging']

}

def deploy():

run('deploy.py')

Remember that Fabric tasks are simply Python functions; they do not necessarily need to execute any code on remote servers. We could then update the staging servers with the following command:

fab staging deploy

This command makes Fabric execute the staging task to set the role definitions and then run the deploy.py script on the remote instance.

###### NOTE

This example runs the nonexistent deploy.py script, which acts as a placeholder for your own deploy script. Later we will create a deploy.py script for the example application from the previous chapter.

That works well when we know all of our hostnames in advance, but how about dynamic fleets of EC2 instances, where we don’t even know how many instances there are, let alone their hostnames?

In combination with the EC2 API, we can take advantage of this feature to selectively execute tasks on our EC2 instances without needing to know the hostnames in advance. This relies on the tagging strategy introduced in the preceding chapters, in which each instance is tagged with a role and an environment. Instead of setting env.roledefs to a list of predefined hostnames, we will query the EC2 API to find a list of instances that matches our target role and environment.

### DYNAMICALLY FINDING INSTANCES

I have released an [open source package](https://github.com/mikery/fabric-ec2) encapsulating the logic required to query the EC2 API and use tags to build up a list of EC2 instance hostnames. This Python module can be used to quickly convert the previous example—which showed how to deploy software to different environments—into a script that can be used to deploy code to all running instances known to EC2.

[Example 7-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch07.html#example_deploy_roles_tags) uses the EC2 tags feature. It assumes that each instance was created with a webor db tag and puts the hostnames into the associated key of an associative array calledroles. For each environment we need (production, staging, and deploy), we read the associative array into our environment.

##### ***Example 7-2. Fabric role definitions and EC2 tags***

from fabric.api import run, sudo, env

from fabric\_ec2 import EC2TagManager

def configure\_roles(environment):

""" Set up the Fabric env.roledefs, using the correct roles for the given environment

"""

tags = EC2TagManager(AWS\_KEY, AWS\_SECRET,

regions=['eu-west-1'])

roles = {}

for role in ['web', 'db']:

roles[role] = tags.get\_instances(role=role, environment=environment)

return roles

def production():

env.roledefs = configure\_roles('production')

def staging():

env.roledefs = configure\_roles('staging')

def deploy():

run('deploy.py')

This can be executed in exactly the same way as the previous static example:

fab staging deploy

Assuming you have some running instances bearing the relevant tags—a host tag with a value of web or db, and an environment tag with the value production—the deployment task will be executed on each of the matching EC2 hosts.

If you wanted to run the deployment task only on the DB instances, as an example, you could execute the following:

fab staging deploy --roles db

The tags given are just examples. Any key/value pairs can be used with EC2 tags and queried from Fabric, making this a flexible method of orchestrating your EC2 fleet.

## Updating Instances at Launch Time

The second part of the problem is to update newly launched instances. If an AMI has a particular version of an application baked into it, that is the version that will be running when the instance is launched. If a new version has been released since the AMI was created, the instances will be running an outdated version.

Our instances therefore need to be able to check for the latest version as part of the boot process and perform an update themselves if necessary. All operating systems provide some mechanism for running user-defined scripts at boot time. For example, on Linux systems, the update can be triggered by placing the following in the /etc/rc.local file:

#!/bin/bash

/usr/local/bin/deploy.py

In this example, the deploy script would need to check a central location to find the latest version of the application and compare it with the currently installed version. The script could then update the instance to the correct version if a change is required.

##### PACKAGE MANAGEMENT

Many programming languages and operating systems offer their own solutions for distributing code and configuration files, such as PyPI for Python, Yum for RedHat-based systems, and Apt for Debian-based systems. Using these systems, where possible, can make for a very easy upgrade path, because you can rely on other aspects of the ecosystem to reduce the amount of work you need to do.

For example, Python’s distribution systems provide a requirements text file that lists all the Python modules required by your application. The requirements file—commonly named requirements.txt—also tracks the installed version of the package. Moving from Boto version 1.1 to 1.5 requires a single change in requirements.txt (from boto==1.1to boto==1.5).

If you package your Python code as a module and publish it to an internal PyPI repository, you can then deploy your application by changing the requirements file used by your running instances.

Another option is to build operating system packages (e.g., RPM packages for RedHat systems) for your custom application so that they can be installed with the OS’s own package management system, and host these on your own repository server. EC2 instances can simply check this repository for any updated packages when they are launched, ensuring that they are always running the correct version of each software package.

Building such systems is not within the scope of this book, but they can be useful for managing AWS-hosted applications, and are well worth the time required to learn and implement.

# AMI-Based Deployments

AMIs are the building blocks of EC2. Deploying AMIs instead of individual code or configuration changes means you can offload the work of replacing instances to Amazon with varying degrees of automation.

## Deploying AMIs with CloudFormation

The most automatic method involves using CloudFormation to replace running instances by changing the AMI ID referenced in the stack template. When CloudFormation receives an update request with a new AMI, it will launch some instances running the new AMI and then terminate the old instances. The number of instances launched will match the number of currently running instances to avoid a sudden reduction in capacity.

Using this method requires a high degree of confidence in the new AMI. You should first test it thoroughly in a staging environment, perhaps with an automated suite of test cases. If the deployment needs to be reverted—going back to the previous AMI—your application will be unavailable until CloudFormation has finished replacing the instances once more.

Furthermore, once CloudFormation starts processing an update, there is no alternative but to wait for it to finish. If you discover early on in the update process that the application is not working as expected, you will need to wait for CloudFormation to finish applying the broken update before issuing a command to perform another update to revert to the previous AMI.

###### TIP

Netflix, a long-term heavy user of AWS, has released [Asgard](http://techblog.netflix.com/2012/06/asgard-web-based-cloud-management-and.html), described as a web-based cloud management and deployment tool. This can be used to automate deployments and changes to your infrastructure and control the process from a web interface. It acts as a supplement to the AWS Management Console and is more tightly integrated with Netflix’s deploy process.

## Deploying AMIs with the EC2 API

An alternative approach is to automate the replacement process yourself using the EC2 API. Instead of allowing CloudFormation to update the running instances, you use the API to perform the same process from your own script. This gives you the opportunity to insert checkpoints and handle the rollback or reversion process with more granularity.

Using the EC2 API opens up some opportunities that were not available in traditional environments. Consider an application that has two web server instances running behind an Elastic Load Balancer, running version 1 of the AMI. The update process could then perform these steps:

1. Launch two instances running version 2 of the AMI.
2. Wait for these instances to be ready.
3. Query the health check page until the instance is serving requests correctly.
4. Add the new instances to the ELB and wait for them to begin receiving traffic.
5. Remove the old instances from the ELB.

At this point, the old instances are still running but not actually serving any traffic. If version 2 of the AMI turns out to be broken, you will see an increase in failed requests as measured by the ELB. You can quickly revert to version 1 of the AMI by performing the update process in reverse—adding the old instances to the ELB and removing the newer ones.

Once you are sure the new version of the AMI is working properly, the old instances can be terminated.

# Recap

The best choice will depend on how frequently you deploy code changes, and how much work is required to create an AMI. Updating running instances is preferable when you are frequently deploying small changes, and you need new code to be live as soon as possible.

Building new AMIs can provide a cleaner way of deploying updates. The new AMI can be thoroughly tested in a staging environment before it is deployed.

The process of deploying an update can be reduced to simply changing the AMI ID used in a CloudFormation stack, and the task of replacing running instances is handled by Amazon’s internal systems. This approach opens up some interesting methods that really take advantage of the cloud and the temporary nature of EC2 instances. The following are some basic principles to take away from this chapter:

* Automate the process of building new AMIs as soon as possible, to reduce deploy friction. The easier it is to deploy changes, the faster you will be able to iterate.
* Do not wait until you have deployed a broken update to start thinking about how to revert.
* AWS has some advantages over physical hardware, such as launching new instances instead of updating code on running instances. Using the EC2 APIs creatively can save both time and headaches.