Jenkins on AWS

AWS Whitepaper

Jenkins on AWS: AWS Whitepaper

Copyright © 2018 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Jenkins on AWS

Publication date: **May 2017** (*Document Details* (p. 30))

Abstract

Although there are many ways to deploy the Jenkins open-source automation server on Amazon Web Services (AWS), this whitepaper focuses on two specific approaches. First, the traditional deployment on top of Amazon Elastic Compute Cloud (Amazon EC2). Second, the containerized deployment that leverages Amazon EC2 Container Service (Amazon ECS). These approaches enable customers to take advantage of the continuous integration/ continuous delivery (CI/CD) capabilities of Jenkins. Using an extensive plugin system, Jenkins offers options for integrating with many AWS services and can morph to fit most use cases (e.g., traditional development pipelines, mobile development, security requirements, etc.).

Introduction

Why CI/CD?

To understand the continuous integration/continuous delivery (CI/CD) model that Jenkins uses, let's start with understanding its underlying drivers. Since the early 2000s, the advent of fast-paced, iterative methodologies such as agile has shifted the thinking about software development. In this new paradigm, product teams push their work to customers as quickly as possible so that they can collect feedback and improve upon the previous iteration of their products. Concepts such as minimum viable product (MVP), release candidate, velocity, etc. are all derived from these new approaches. In contrast, product teams using older paradigms like waterfall development might not hear back from customers for months and, quite often, not until the product is commercialized.

The following figure illustrates the high-level phases of product development:

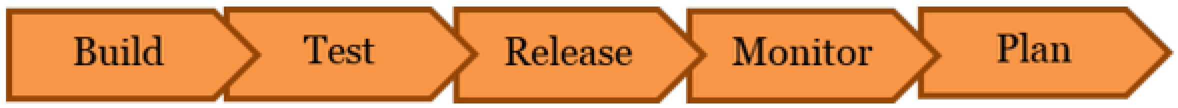


Figure: High-level product development phases

The order and length of these phases varies depending on which development models is used (e.g., waterfall, v-model, scrum, etc.).

Continuous Integration

Continuous integration (CI) is a software development practice in which developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g., a CI or build service) and a cultural component (e.g., learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

The basic challenges of continuous integration include maintaining a single source code repository, automating builds (and building fast), and automating testing. Additional challenges include testing on a clone of the production environment, providing visibility of the process to the team, and allowing developers to obtain the latest version easily. The goal of this whitepaper is to show you how using Jenkins on AWS is a strategy fit to address these CI challenges.

Continuous Delivery and Deployment

Continuous delivery (CD) is a software development practice where code changes are automatically built, tested, and prepared for production release. It expands upon continuous integration by deploying all code changes to a testing environment, a production environment, or both after the build stage has been completed. When continuous delivery is properly implemented, developers always have a deployment-ready build artifact that has passed through a standardized test process. With continuous deployment, revisions are deployed to a production environment automatically without explicit approval from a developer, making the entire software release process automated. This, in turn, allows for the product to be in front of its customers early on, and for feedback to start coming back to the development teams.

Why Use Jenkins?

Jenkins is a very popular product among AWS customers who want to automate their CI/CD pipelines.

- It accomplishes all of the phases described in the previous section.
- It integrates very well across languages, platforms, and operating systems.
- It’s open-source software.

Jenkins works well on AWS and *with* AWS because it’s available on the AWS Marketplace, it’s widely documented, and it’s very well integrated. Additionally, Jenkins plugins are available for a number of AWS services. The rest of this whitepaper discusses some of those plugins and what they allow our customers to accomplish.

> **From the Jenkins official website:**
> "The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project."
> https://jenkins.io/

Deploying Jenkins on AWS

In this section we discuss two approaches to deploying Jenkins on AWS. First, you could use the traditional deployment on top of Amazon Elastic Compute Cloud (Amazon EC2). Second, you could use the containerized deployment that leverages Amazon EC2 Container Service (Amazon ECS). Both approaches are production-ready for an enterprise environment. In addition, both approaches place the Jenkins environment inside an Amazon Virtual Private Cloud (Amazon VPC).

Topics

Amazon VPC

Amazon VPC lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.

We highly recommend that you launch your Jenkins installation within a VPC. Launching inside a VPC not only allows you to keep your Jenkins resources separate from other resources you might be running, but also provides the ability to have control over static instance IP addresses, which will be important as you add worker nodes to your configuration (more on that later).

Traditional Deployment

A Jenkins traditional deployment means deploying Jenkins on top of Amazon EC2. Later in this whitepaper we explore containerized deployment.

Overview of Jenkins Architecture

The Jenkins architecture is fairly straightforward. Out of the box, it’s deployed as both a server and a build agent running on the same host. You can choose to deploy Jenkins as either a server or a build agent, which allows for decoupling orchestration and build execution. This, in turn, allows for more architecture design flexibility.

Strategies for Jenkins Worker Node Deployments

By default, the Jenkins server will handle all HTTP requests as well as the builds for each project. As the number of users grows, or the amount or complexity of jobs increases, the master server may experience degraded performance due to a taxing of resources like CPU and memory, or due to the number of builds that are running on the master server.

This is when build agents (or *worker nodes*) can benefit a Jenkins installation by freeing up resources on the master node and providing customized environments in which to test builds. A worker node contains an agent that communicates with the master server and runs a lightweight Jenkins build that allows it to receive and run offloaded jobs.

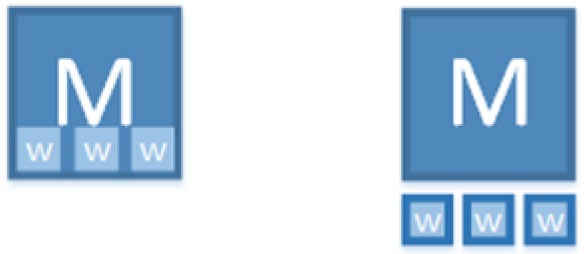


Figure: Master and Worker deployment options

Strategies for Jenkins Master Deployments

Jenkins installations generally fall into one of two scenarios:

1. A single, large master server with multiple worker nodes connected to it.
2. Multiple smaller master servers with multiple worker nodes connected to each.

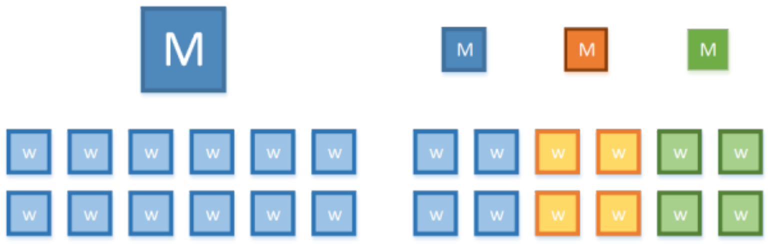


Figure: Jenkins deployment strategies

In both cases, one or more worker nodes are present. In larger systems, this is an important practice—do not build on the master. Choosing between a single master or multiple masters depends on a few factors, but usually we see customers adopt multiple masters. For example, Netflix runs more than 25 masters on AWS.

The following table provides some criteria that you can use when you're choosing which strategy best fits your needs:

	Single Master Strategy	**Multi Master Strategy**
Number of teams	Few	Many
Plugins	Consistent set across all teams	Varied set across teams
Custom configurations per team	Harder to manage	Easier to manage
Server maintenance	Easier to manage	Harder to manage

While there are many successful strategies for deploying multiple Jenkins masters, this paper focuses on deploying a single Jenkins master to Amazon EC2, but architecting it to be self-healing and highly available. The techniques described here for deploying a single Jenkins master can be employed to create a multiple-master environment.

Architecting for High Availability

The AWS infrastructure in the following figure is built around Regions and Availability Zones (AZs). A Region is a physical area of the world where AWS has multiple Availability Zones. An Availability Zone consists of one or more discrete data centers, each with redundant power, networking, and connectivity, housed in separate facilities. These Availability Zones offer you the ability to operate production applications and databases that are more highly available, fault tolerant, and scalable than would be possible from a single data center.

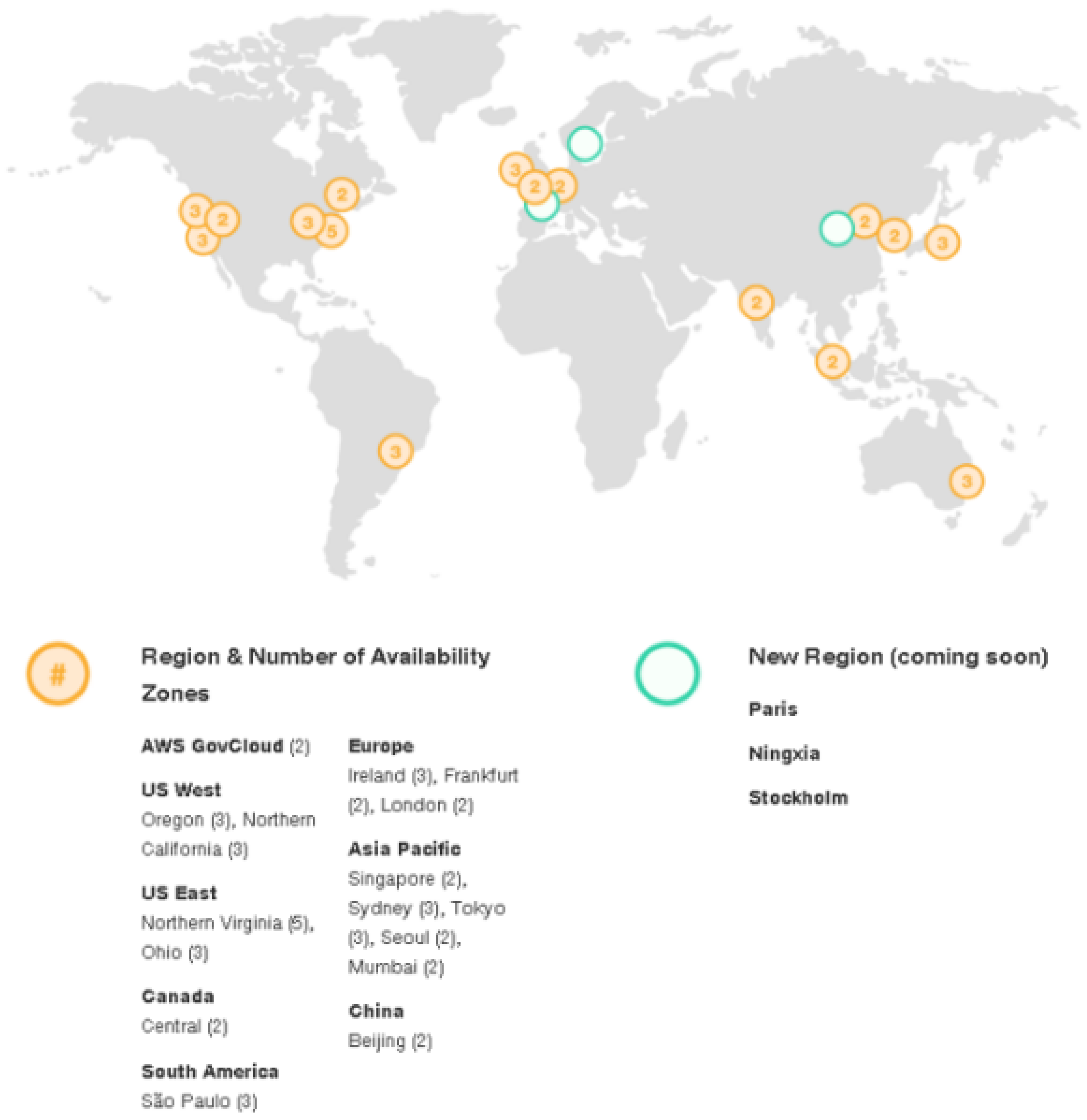

3
2
3
2
3
5
3
2
2
2
2
3
2
2
3
3
#
Region & Number of Availability Zones
AWS GovCloud (2)
US West
Oregon (3), Northern California (3)
US East
Northern Virginia (5), Ohio (3)
Canada
Central (2)
South America
São Paulo (3)
Europe
Ireland (3), Frankfurt (2), London (2)
Asia Pacific
Singapore (2), Sydney (3), Tokyo (3), Seoul (2), Mumbai (2)
China
Beijing (2)
New Region (coming soon)
Paris
Ningxia
Stockholm

Figure: Current AWS global infrastructure

In the AWS Cloud, a web-accessible application like Jenkins is typically designed for high availability and fault tolerance by spreading instances across multiple Availability Zones and fronting them with an Elastic Load Balancing (ELB) load balancer. Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances in the cloud. It enables you to achieve greater levels of fault tolerance in your applications and seamlessly provides the required amount of load balancing capacity needed to distribute application traffic.

Due to the fact that Jenkins stores master node configuration in the `$JENKINS_HOME` directory—rather than, say, a database—it becomes problematic to maintain a redundant Jenkins server in a separate Availability Zone in a single master setup without the aid of plugins. Tasks like configuring Jenkins, maintaining plugins, and managing users would need to be repeated across each replicated instance in each Availability Zone.

Note
You should explore the use of plugins. There are plugins available to make your Jenkins environment more effective and useful.
For example, using a plugin like the High Availability plugin from CloudBees, you could set up the $JENKINS_HOME directory on a shared network drive, so it could be accessible by multiple Jenkins servers behind and ELB load balancer. This would provide a fault-tolerant environment. The Private SaaS Edition by CloudBees is another option.

If your business requirements demand a fault-tolerant Jenkins environment, your preferred setup might be a scenario in which multiple masters with their own workers are placed in separate Availability Zones.

Because the focus of this whitepaper is on the single master scenario, you should consider creating an Amazon CloudWatch alarm that monitors your Jenkins instance. CloudWatch automatically recovers the instance if it becomes impaired due to an underlying hardware failure or a problem that requires AWS involvement to repair. This method gives you the ability to quickly recover in the event of a failure without having the benefit of running Jenkins across multiple Availability Zones.

Later in this whitepaper, we propose that decoupling your configuration storage from the Jenkins compute node allows for increased availability.

These options do not require additional Jenkins plugins. Instead they rely on AWS technology.

Resource Considerations for Right-Sizing Your Jenkins Master

As with any AWS deployment, sizing your instance in terms of CPU, memory, and storage has an impact on performance and cost profiles. It's therefore very important to make the right choices.

Operating System

We discuss a Jenkins deployment using Amazon Linux, but it's easily adapted to other flavors of Linux or to Windows environments using the Windows installer for Jenkins.

CPU and Networking

A Jenkins deployment built for scale and high availability varies based on the number of worker nodes that connect to a master node. A master Jenkins node launches multiple threads per connection—two for each SSH and Java Web Start (JWS) connection and three for each HTTP connection.

Jenkins serves dashboard content over HTTP. Therefore, if you expect a large number of concurrent users, you should expect additional CPU overhead for the rendering of this content.

Storage

A Jenkins deployment for large teams should ensure that worker nodes perform the build. Therefore it's more important to have large storage volumes on the worker nodes than on the master. Jenkins settings and build logs are stored on the master by default, so when you plan for volume size on your master consider the number and size of the projects you will maintain. In this installation, storage in the form of Amazon Elastic Block Store (Amazon EBS) volumes is attached to both the primary and worker node instances. Amazon EBS provides persistent, block-level storage volumes to deliver low-latency and consistent I/O performance scaled to the needs of your application.

EBS volumes are designed for an annual failure rate (AFR) of between 0.1% - 0.2%, where failure refers to a complete or partial loss of the volume, depending on the size and performance of the volume. This makes EBS volumes 20 times more reliable than typical commodity disk drives, which fail with an AFR of around 4%.

Instance Type

When building your Jenkins environment on Amazon EC2, consider CPU, networking, and storage. We encourage you to benchmark your project builds on several Amazon EC2 instance types in order to select the most appropriate configuration.

We benchmarked five different instance types in our evaluations: the t2.large, the m3.medium, and the m4.large, m4.xlarge, and m4.2xlarge. Each benchmark simulated traffic from 100 concurrent users loading multiple pages inside the Jenkins dashboard for a sustained period of 10 minutes.

Overall, we found the m4.large to be the best value for the performance. The average CPU utilization during load testing did not exceed 3%, with an average response time of 220 milliseconds. As expected, the m4.xlarge and m4.2xlarge sizes performed well but at a higher cost per hour; therefore, the m4.large remains the best choice for our needs.

The m3.medium, while a good choice for many applications, did not perform as well as the m4.large and had an average CPU utilization of over 80% for the duration of the testing.

The t2.large performed well during the first few minutes of testing. However, because t2 instances offer burstable performance, a sustained amount of high traffic from 100 users depleted available CPU credits, and performance significantly decreased. Further testing with fewer users (i.e., 10 users) saw improved results. Thus, if you have a relatively small team and do not expect frequent or high-volume usage from your Jenkins master, the T2 family may be a good option for you.

In this whitepaper, we build a Jenkins master using the m4.large, which comes with 2 vCPUs and 8 GB of memory. A general purpose solid-state drive (SSD) EBS volume of 20 GB is provisioned and attached to the instance. General purpose SSD volumes are the default EBS volume type for Amazon EC2 and are backed by SSDs, making them suitable for a broad range of workloads, including small- to medium-sized databases, development and test environments, and boot volumes.

Installation

To install the Jenkins Master Node, do the following.

Master Node

1. From the AWS Management Console, launch the Amazon EC2 instance from an Amazon Machine Image (AMI) that has the base operating system you want. This example uses an Amazon Linux 64-bit AMI.
2. Choose a security group that will allow SSH access as well as port 80 or 8080 to access your Jenkins dashboard. You should only enable ingress from the IP addresses you wish to allow access to your server.
3. Connect to the instance via SSH.
4. Update the yum package management tool.

```
$ sudo yum update -y
```

5. Download the latest Jenkins code package.

```
$ sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins.io/redhat/jenkins.repo
```

6. Import a key file from Jenkins-CI to enable installation from the package.

```
$ sudo rpm --import https://pkg.jenkins.io/redhat/jenkins.io.key
```

7. Install Jenkins.

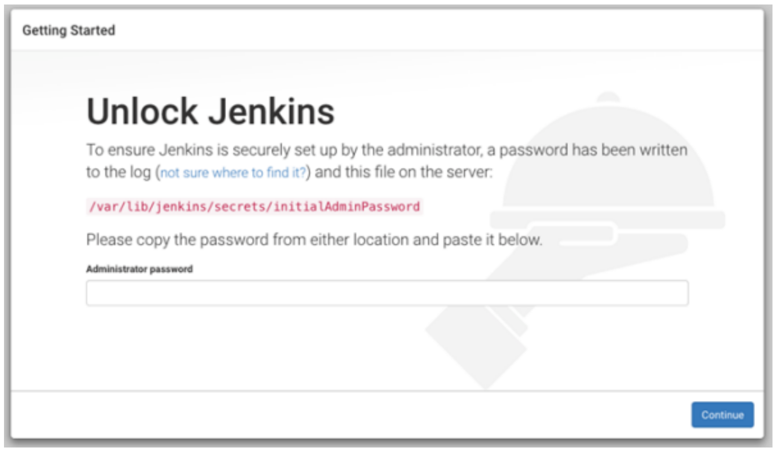
```
$ sudo yum install jenkins -y
```

8. Start Jenkins as a service.

```
$ sudo service jenkins start
```

9. Configure Jenkins, now that it's installed and running on your Amazon EC2 instance. Use its management interface at port 80 or 8080, or remotely access the server via SSH. In its default configuration, Jenkins versions 2.0 and later lock down access to the management interface.

   The first time you use the dashboard at `http://your-server-address:8080`, you will be prompted to unlock Jenkins:


Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

/var/lib/jenkins/secrets/initialAdminPassword

Please copy the password from either location and paste it below.

Administrator password

Figure: Unlock Jenkins

   As noted on the user interface, you can find this password in `/var/lib/jenkins/secrets/initialAdminPassword`. Paste the value into the password box, then choose **Continue**.

10. The installation script directs you to the Customize Jenkins page. Choose **Select plugins to install** and select any plugins appropriate to your particular installation. For our example, ensure that the Git plugin (under Source Code Management) and SSH Slaves plugin (under Distributed Builds) are installed.

Security Considerations

At a minimum, incoming traffic to the Jenkins master should be locked down to the specific IP address ranges from which you expect traffic. Additionally, your environment can be further secured using the following methods.

Enable SSL

Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates can be used to secure network communications and establish the identity of websites over the Internet. You can make this easy by fronting your Jenkins master with the ELB load balancer. In that case, you can use the AWS Certificate Manager to easily provision a certificate that gives you encrypted network connections and protects your data as it travels across the wire.

CSRF Protection

Cross-site request forgery (CSRF) is a class of attack that forces an end user to execute unwanted actions on Jenkins. By default, Jenkins 2.x installations have the CSRF protection option enabled. To check the status of this setting, choose **Manage Jenkins**, then **Configure Global Security**, and ensure that Prevent Cross-Site Request Forgery Exploits is enabled.

Security Implication of Building on Master

Care should be taken not to perform builds on the master. As you see in the following configuration steps, we recommend that you configure the master to have no executors, and that you run builds only on worker nodes. Builds that are run on the master have the ability to read and modify files in `$JENKINS_HOME`, which, if accessed maliciously, can affect the entire Jenkins installation and the security of your system.

Worker Node Access Control

Jenkins versions 1.5.80.x and later have a subsystem in place that establishes a barrier between worker nodes and a master to safely allow less trusted worker nodes to be connected to a master.

Your Jenkins 2.x installation should have this barrier enabled by default. You can verify this by visiting “Manage Jenkins,” then “Configure Global Security,” and ensuring that “Enable Slave->Master Access Control” is enabled.

Configure User Authentication

During the installation process, you can choose to create a First Admin User. Create a master user that can be used to create other groups and users, and then proceed to the Jenkins dashboard.

User authentication can be provided through a number of methods:

- The simplest authentication scheme is to use the Jenkins user database. User accounts can be created using the Jenkins dashboard (choose **Manage Jenkins**, then **Configure Global Security**).
    - The option for Matrix-based security offers the most precise control over user privileges. Using this option you can specify fine-grained permissions for each user and each action they can take.
- If Jenkins is running on a Windows machine, you can configure Jenkins to authenticate the user name and the password through Active Directory using the Active Directory plugin.
- If Jenkins is running on Linux, the same Active Directory plugin can be used by specifying the Active Directory domains to authenticate with, or you can configure access to Unix users/groups in your local system. With this setting, users will be logged into Jenkins by entering their Unix username and password.
- With the LDAP plugin, users can authenticate with an Active Directory-compliant LDAP service such as AWS Directory Service or OpenLDAP.

Securing Network Access

A security group acts as a virtual firewall that controls traffic to your instances. When you launch an instance, you associate one or more security groups with the instance. Because all ports are disabled by default, you add rules to each security group that allow traffic to or from its associated instances.

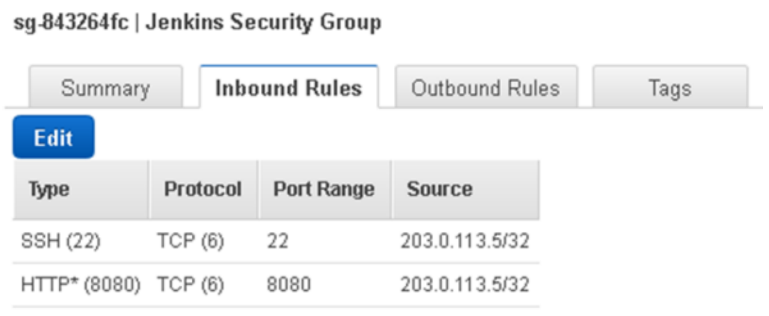
When launching your master, create a security group that allows ingress to the following ports:

1. Port 80 or 8080, for the ability to configure Jenkins and interact with the management dashboard. By default, the Jenkins dashboard is accessible through port 8080, but you can use iptables to redirect port 80 to 8080 and allow local connections:

```
$ sudo iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8080
$ sudo iptables -t nat -I OUTPUT -p tcp -o lo --dport 80 -j REDIRECT --to-ports 8080
```

2. Port 22, to connect via an SSH connection to the instances and perform maintenance.

For each port, restrict access to your IP address or an IP address range using the Source field, which determines the traffic that can reach your instance. Specify a single IP address or an IP address range in classless inter-domain routing (CIDR) notation (e.g., 203.0.113.5/32 as in the following figure). If connecting from behind a firewall, you'll need the IP address range used by the client computers.



Type	Protocol	Port Range	Source
SSH (22)	TCP (6)	22	203.0.113.5/32
HTTP* (8080)	TCP (6)	8080	203.0.113.5/32

Figure: Security group configuration

Worker Nodes

A Jenkins worker node is an instance that offloads build projects from the master. The Jenkins master makes the distribution of tasks fairly automatic and is configurable per project. Users accessing Jenkins using the management dashboard can track build progress, browse Javadoc files, and view and download test results without needing to be aware that builds were done on worker nodes.

Each worker node runs a worker agent, which eliminates the need to install the full Jenkins package on those instances. These agents can be installed using various methods, each with the end result of establishing bi-directional communication between the master instance and the worker instance.

After you choose an instance type and resources for your worker nodes, it’s best to build them to be general purpose, rather than building them for a specific project. This is especially true on large teams or in environments with a variety of different projects. As a rule of thumb, remember that your worker nodes should be fungible—easily replaced by another node in the event of failure.

Note
You can significantly reduce your costs using Amazon EC2 Spot Instances to run workers. Amazon EC2 Spot Instances allow you to bid on spare EC2 computing capacity and achieve up to 90% cost savings. Spot Instances are particularly adapted for asynchronous workloads such as those that Jenkins workers run. To take advantage of Spot instances, configure Jenkins to use the Amazon EC2 Fleet Plugin.

> Lyft, an AWS customer, was able to save 75% a month using Spot instances. For more information, see the Lyft case study.

After configuring a suitable worker node and installing your necessary packages and tools (e.g., Maven or Git), create an AMI from which you can quickly launch other nodes.

The worker nodes will only need port 22 to be open for communication with a master. They will not need port 8080 open. Create a security group that allows ingress to port 22, and restrict access to the IP address of both your master node and your specific IP address on your internal network.

You should benchmark performance for your projects to determine the appropriate instance type and size you need, as well as the amount of disk space. In this whitepaper, we specify that our Jenkins worker nodes use an m4.large instance type, which comes with 2 vCPUs and 8 GB of memory, along with a 40 GB general purpose SSD Amazon EBS volume.

Worker nodes connect to the master in a variety of different ways—via SSH, via Windows through the remote management facility (built into Windows 2000 or later), via Java Web Start, or via a custom script.

For Linux installations, SSH is the most convenient and preferred method for launching worker agents, and is the one used in this whitepaper.

Installation

1. Launch a new instance into the same VPC that you launched the Jenkins master into. For this example, we launch the instance from an Amazon Linux 64-bit AMI. (Note that you will not have to install Jenkins on your workers.)
2. Attach a security group that allows ingress to the worker nodes via SSH from the internal IP address of the master Jenkins install.
3. Connect to the master instance via SSH.
4. Create a pair of authentication keys from the ~/.ssh directory:

```
$ cd ~/.ssh
$ ssh-keygen -t rsa
```

5. Copy the public key—you'll need it in step 7.

```
$ cat ~/.ssh/id_rsa.pub
```

6. From another terminal window, connect to the worker instance via SSH.
7. Edit `~/.ssh/authorized` keys and add the contents of the public key from step 5 to the end of the file. Take care not to include any line breaks that you may have copied as you captured the public key.
8. Return to the terminal where you connected to the master instance via SSH, and verify that you can connect to the worker instance the same way.

```
$ ssh ec2-user@111.222.333.444
```

Configuration

On the master instance, log in to the Jenkins management dashboard, choose **Manage Jenkins**, then **Manage Nodes**, and finally **New Node**. For each worker node, perform the following steps:

1. Name your node and choose **Permanent Agent**. Then choose **OK** to continue. (If you have already configured a worker node, you can choose **Copy Existing Node** to duplicate a previous configuration.)

2. Add a description and set the number of executors you wish to run on the worker node. Plan for two executors per core.
3. Set **Remote Root Directory** to an absolute path, such as `/home/ec2-user/jenkins`. The remote root directory will cache data such as tool installations or build workspaces. This prevents unnecessary downloading of tools, or checking out source code again when builds start to run on this agent again after a reboot. Set any other options you want.
4. For the launch method, choose **Launch slave agents on Unix machines via SSH** and set the host to the private IP address of your worker node.
5. Add your credentials by choosing **Add**, then **SSH Username with private key**.
6. Add the value "ec2-user" for **Username**.
7. For Private Key, choose **Enter directly**, then paste in the value of the private key (`~/.ssh/id_rsa`) from the master instance, and choose **Add**. Finally, choose **Save** to save your worker node configuration.
8. Select the worker from the nodes table, and then choose **Launch agent**.

After you have configured your worker nodes, it's important to change the number of executors to 0 on the master. Select the master from the nodes table, and configure the "# of executors" settings to 0. This ensures that project builds will now happen on the worker nodes, and not on the master.

Test your Jenkins setup by adding a new project and performing a build.

Decoupling Configuration from Our Jenkins Installation

In the previous section we hinted that options for highly available (HA) deployments are limited, with the exception of the deployments that CloudBees has built.

This section illustrates how decoupling those configuration files from the machine running the Jenkins master node allows for an increased level of reliability and availability. We will rely on Amazon Elastic File System (Amazon EFS) to store `$JENKINS_HOME`, and we will implement Auto Scaling launch configurations to ensure that our installation can recover across Availability Zones, if anything happens to our Jenkins installation.

Create an Amazon EFS file system

1. In the AWS Management Console, go to the Amazon EFS dashboard.
2. Choose **Create File System**.
3. Select the target VPC into which you want to deploy your file system. This should be the same VPC where you put your Jenkins setup.
4. Under **Create mount targets** configure the mount targets you need. Instances connect to a file system via mount targets you create. We recommend creating a mount target in each of your VPC's Availability Zones so that Amazon EC2 instances across your VPC can access the file system.

   When choosing a security group, ensure that Amazon EFS allows traffic from your Jenkins instances. These security groups act as a virtual firewall that controls the traffic between them. If you don't provide a security group when creating a mount target, Amazon EFS associates the default security group of the VPC with it. That group allows all traffic across instances so it will server our purpose.

   Regardless, to enable traffic between an EC2 instance and a mount target (and thus the file system), you must configure the following rules in these security groups:

   a. The security groups you associate with a mount target must allow inbound access for the TCP protocol on port 2049 for NFS from all EC2 instances on which you want to mount the file system.

   b. Each EC2 instance that mounts the file system must have a security group that allows outbound access to the mount target on TCP port 2049.
5. Choose **Next Step**.
6. Add tags if you need them, and choose performance mode. We recommend General Purpose performance mode for most file systems. Max I/O performance mode is optimized for applications where tens, hundreds, or thousands of EC2 instances are accessing the file system — it scales to higher levels of aggregate throughput and operations per second with a tradeoff of slightly higher latencies for file operations.
7. Choose **Next Step**.
8. Review your configuration and then choose **Create File System**.

Setting Up Your Jenkins Host

1. Launch an instance into your VPC from an AMI (our example uses an Amazon Linux 64-bit AMI). Follow the instructions given earlier, and be sure to pick a security group that allows SSH and HTTPS (ports 22 and 8080, respectively).
2. Connect to the instance via SSH.
3. Update the yum package management tool:

```
$ sudo yum update -y
```

4. Install nfs-utils, if necessary:

```
$ sudo yum install nfs-utils
```

   If you choose the Amazon Linux AMI 2016.03.0 when launching your EC2 instance, you won't need to install nfs-utils because it's already included in the AMI by default.
5. Create a folder for mounting your `$JENKINS_HOME` folder:

```
$ sudo mkdir -p /mnt/JENKINS_HOME
```

6. Mount the Amazon EFS file system to this directory. Use the following command, replacing *`file-system-id`* and *`aws-region`* placeholders with your file system ID and AWS Region, respectively:

```
$ sudo mount -t nfs4 -o vers=4.1
$(curl -s http://169.254.169.254/latest/meta-data/placement/availability-zone).file-
system-id.efs.aws-region.amazonaws.com://mnt/JENKINS_HOME
```

   Tip
   `aws-region` will be one of the following values:

Region Name	*`aws-region`*
US East (N. Virginia)	us-east-1
US East (Ohio)	us-east-2
US West (Oregon)	us-west-2
EU (Ireland)	eu-west-1
Asia Pacific (Sydney)	ap-southeast-2

7. Install Jenkins:

```
$ sudo wget -O /etc/yum.repos.d/jenkins.repo
http://pkg.jenkins-ci.org/redhat/jenkins.repo
$ sudo rpm --import
https://pkg.jenkins.io/redhat/jenkins.io.key
$ sudo yum install jenkins -y
```

8. Change the ownership of the newly created mount:

```
$ sudo chown jenkins:jenkins /mnt/JENKINS_HOME
```

9. Update Jenkins configuration:

```
$ sudo vi /etc/sysconfig/jenkins
```

   Inside that file, replace the value of *`$JENKINS_HOME`* with *`/mnt/JENKINS_HOME`*. If there are already configuration files in the existing `$JENKINS_HOME` directory (`/var/lib/jenkins` by default), make sure to move them to the new `/mnt/JENKINS_HOME` directory.

10. Launch Jenkins:

```
$ sudo service jenkins start
```

11. Make sure the share auto-mounts when the box starts up:

```
$ sudo vi /etc/fstab
```

   Add the line:

```
mount-target-DNS:/ /mnt/JENKINS_HOME nfsdefaults,vers=4.1 0 0
```

   Replace *`mount-target-DNS`* with the DNS name of your Amazon EFS server. (You used that DNS name when mounting the Amazon EFS volume on the machine earlier, in Step 6.)

Setting Up an Auto Scaling Group for Auto-recovery

At this point you have a Jenkins server that relies on Amazon EFS for its configuration. This means that if your Jenkins server disappeared, you could replace it with another Jenkins install pointing to the Amazon EFS share where your data is stored.

You could certainly replace that server manually (rerunning the steps above). However, we want that to happen automatically, relying on standard AWS features.

1. Create an AMI of your instance:
   a. On the AWS Management Console, right-click your instance.
   b. Choose **Image->Create Image**, give it a name, choose **Create Image**, and wait for a few minutes.
2. Create a launch configuration:
   a. On the Amazon EC2 Dashboard, choose **Launch Configuration** in the navigation pane at the left, then choose **Create Auto Scaling Group**. (The console might show a **Create launch configuration** button if you already have some Auto Scaling groups. Choose that button instead if that’s the case.)
   b. Choose **Create launch configuration**.
   c. Pick the AMI you just created. (Choose **My AMIs** in the navigation pane to show your AMIs.)

   d. Fill out the required information over the next few pages. Be sure to select the security group you created earlier. (Ports 22 and 8080 should be open.)
   e. Finally, choose **Create launch configuration** from the Review page and select a key you have access to.
3. Create an Auto Scaling group:
   a. Before running the steps below, shut down your existing, manually created Jenkins instance so that it doesn't interfere with the ones that will be spun up as part of your Auto Scaling group.
   b. On the Amazon EC2 Dashboard, choose **Auto Scaling Group** in the navigation pane, then choose **Create Auto Scaling group**.
   c. Pick the Launch Configuration you just created and choose **Next Step**.
   d. Name your Auto Scaling group and set Group Size = 1. (Remember, we can only run one instance at a time.)
   e. Pick the proper VPC and subnets. (Be sure to select subnets that are on different Availability Zones.) Choose **Next: Configure scaling policies**.
   f. Do not use Scaling Policies so that the group stays at its initial size.
   g. Click through the next few pages, and finally choose **Create Auto Scaling group** on the Review page.

Expansions

At this point, if you manually shut down your Jenkins server, Auto Scaling starts a new one that will pick up where the old server left off. Consider also doing the following:

- Set up a load balancer pointing to your Auto Scaling group so that you can consistently find your server using the load balancer DNS name. (As your server gets replaced, its IP address will change.)
- As mentioned before, use Amazon EFS or any other distributed and highly reliable storage mechanism.
- Capture this setup in an AWS CloudFormation template to make it easy to reproduce.

Best practices

The following are best practices for a traditional deployment.

Security

In its default configuration, Jenkins doesn't perform any security checks that could expose passwords, certificates, and private data, as well as leave job builds and configurations open to anyone. Using security groups, configure user security and network security on both your master instance and work nodes to limit the vectors at which an attacker can compromise your system.

Instances

Launch your Jenkins master from an instance that has enough CPU and network bandwidth to handle concurrent users. Configure worker nodes so that they are identical to each other. They should run on the same instance family from the same generation, and builds should happen on the worker nodes, not the master. Worker nodes should be fungible—able to be thrown away quickly, and brought up or added into the cluster with as little manual interaction as possible. Use AMIs as described earlier to create a default image for your worker nodes. Then, launch additional worker nodes as necessary based on this image.

If your teams build at dedicated or predictable times, you can stop worker nodes when jobs are not running, and turn them on only when you need them. This way you don't pay for idle instances.

Monitoring

At all times, monitor your instances, especially CPU and disk performance. Take advantage of Amazon CloudWatch alarms to monitor instance resources like CPU usage or disk utilization. This will help you right-size your instances and volumes. Email and SMS alerts can be configured to immediately notify you when events like low disk space or high CPU utilization cross a threshold that you define.

Backup and Restoration

Maintaining a regular backup of your Jenkins master is crucial to providing a stable environment. A backup ensures that your Jenkins instance can be restored in the event of data corruption or loss, or misconfiguration of Jenkins, which leaves it in a usable state.

You can perform a backup by either taking a snapshot of the entire server or by backing up the `$JENKINS_HOME` directory.

Amazon EBS provides a feature for backing up the data on your Amazon EBS volumes to Amazon Simple Storage Service (Amazon S3) by taking point-in-time snapshots. We strongly recommend that you take regular snapshots of the Amazon EBS volumes backing your Jenkins master. Because you can launch a new volume based on a snapshot, you can quickly recover in the event of a failure.

Instead of taking a snapshot of the entire volume, you can choose to just back up the `$JENKINS_HOME` directory, which contains your Jenkins-specific configurations. When you restore, you simply launch a new Jenkins master and replace the `$JENKINS_HOME` directory with the contents of your backup.

Finally, there are many plugins available to manage backups for your `$JENKINS_HOME` directory, such as the S3 Plugin, which backs up your configuration to Amazon S3, which is designed to provide 99.999999999% durability and 99.99% availability.

Further Reading

For more information on best practices for using Jenkins in your projects' lifecycles, we encourage you to read the Jenkins Best Practices wiki.

Containerized Deployment

Containers are a form of operating system virtualization that allow you to run an application and its dependencies in resource-isolated processes.

Overview of Container Technology

Containers allow you to easily package an application's code, configurations, and dependencies into a template called an *image,* which is used to launch the container. Containers can help ensure that applications deploy quickly, reliably, and consistently regardless of the deployment environment. You also have more granular control over resources, which can improve the efficiency of your infrastructure.

Overview of Container-based Jenkins

Another approach to deploying and running Jenkins is to run the master and the worker nodes in Docker containers. A Docker container packages an application, its configurations, and all its dependencies into a single unit of deployment that contains everything the application needs to run.

Running Jenkins in Docker containers allows you to use servers running Jenkins worker nodes more efficiently. It also simplifies the configuration of the worker node servers. Using containers to manage builds allows the underlying servers to be pooled into a cluster. The Jenkins worker containers can then run and execute a build on any of the servers with resources available to support the build. This ability to pool multiple builds to run independently of each other on the server improves the utilization of the server.

Another benefit of containers is that they allow for tests to be run in “clean” environments *every* time, as opposed to environments that can potentially become “dirty” over time and corrupt the tests.

Configuration is also simplified when you use containers for running the workers. The same base configuration can be reused multiple times. By making the worker containers fungible you can easily run multiple compiler types and versions, libraries, and any other dependencies.

Amazon ECS

Amazon EC2 Container Service (Amazon ECS) is a container management service that supports creating a cluster of Amazon EC2 instances for running the Jenkins master nodes and worker nodes as Docker containers. This cluster of servers is typically referred to as a *build farm*.

Amazon ECS eliminates the need for you to install, operate and scale your own cluster management infrastructure. You can use API calls to launch and stop Docker-enabled applications, query the complete state of your cluster, and access many familiar features like security groups, load balancers, Amazon EBS volumes, and IAM roles.

The build farm based on Amazon ECS consists of the following components:

- Cluster – A logical grouping of container instances into which you can place Amazon ECS tasks for the Jenkins master or worker node.
- Amazon ECS Container Agent – Allows container instances to connect to the cluster and is included in the AMI that is optimized for Amazon ECS. The Amazon ECS container agent can also be installed on any EC2 instance that supports the Amazon ECS specification.
- Container instance – The EC2 instance that is running the Amazon ECS container agent and has been registered into a cluster. The Jenkins master and worker nodes can run on any of the container instances in the cluster.
- Task definition – An application description that contains one or more container definitions. The task definition captures how the task running the Jenkins master is configured, for example, how much memory the Jenkins master needs, what port it would be accessed on, etc.
- Scheduler – The method used for placing tasks on container instances. The scheduler determines what container instance a Jenkins worker node or master runs on when it’s started.
- Service – Allows you to run and maintain a specified number of instances of a task definition simultaneously. The Jenkins master is a long-running task and runs as a service.
- Task – An instantiation of a task definition running on a container instance. The Jenkins worker nodes run as tasks.
- Container – A Docker container created as part of a task.

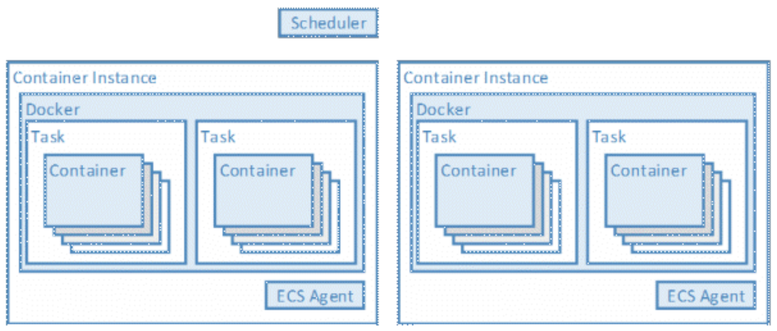


Figure: Amazon ECS components

Implementation

In this section we review the steps needed to implement containerized Jenkins.

Setting Up the Amazon ECS Cluster for Our Build Farm

The first step in setting up the build farm is to launch an Amazon ECS cluster in a VPC. You must specify the instance type to be used for the build servers and the number of Amazon EC2 instances needed in the build farm.

Creating the Jenkins Master Service

To create the Jenkins master, create a Dockerfile that uses the official Jenkins Docker image as the base. The Dockerfile should also contain the steps to install the Jenkins Amazon ECS plugin and any other plugins needed. Specifically, the Amazon ECS plugin will allow the Jenkins master to use Amazon ECS to orchestrate starting up and tearing down the Jenkins workers as needed.

```
# Dockerfile for Jenkins Master
FROM Jenkins
# Add the entry amazon-ecs to plugin.txt to preload the Amazon ECS plugin
COPY plugins.txt /usr/share/jenkins/plugins.txt
RUN /usr/local/bin/plugins.sh /usr/share/jenkins/plugins.txt
```

Use the Dockerfile to create an image and push the image to the Amazon EC2 Container Registry (Amazon ECR). Amazon ECR is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. The custom image can now be used to run one or more Jenkins masters in the Amazon ECS cluster. (You might do this if, for example, you have several teams that each need their own Jenkins environment.)

```
# Login using docker command returned by the command below
aws ecr get-login --region region
# Build your Docker image using the Dockerfile
docker build -t jenkins-master .
# Tag the jenkins_master image
docker tag jenkins_master:latest AWS Account Number.dkr.ecr.us-east-1.amazonaws.com/
jenkins_master:latest
# Push the jenkins-master image to ECR
docker push AWS Account Number.dkr.ecr.us-east-1.amazonaws.com/jenkins-master:latest
```

Jenkins uses the `JENKINS_HOME` directory to store information on installed plugins, logs, build jobs, and other configuration settings. By design, the data in the Jenkins master container is ephemeral and is lost when the container is stopped or restarted. If you don't want to lose this data, Amazon ECS supports persisting the data by using data volume containers.

The Jenkins master container persists the data in `JENKINS_HOME` by mounting a volume from the data volume container.

Behind the scenes, the data volume in the data container is associated with a directory in a file system in an Amazon EBS volume attached to the container instance.

Note that Amazon ECS does not sync your data volume across container instances. The Jenkins master container has to be started on the same container instance that has the data container volume.

To create a data volume container for use by the Jenkins master container, create a Dockerfile that exports the Jenkin home directory `/var/jenkins_home`.

```
# Dockerfile for Jenkins Data Volume Container
FROM Jenkins
VOLUME ["/var/jenkins_home"]
```

```
# Creating a Jenkins Data Volume
docker build -t jenkins_dv .
#Tag the jenkins_dv image
docker tag jenkins_dv:latest AWS Account Number.dkr.ecr.us-east-1.amazonaws.com/
jenkins_dv:latest
# Push the jenkins-dv image to ECR
docker push AWS Account Number.dkr.ecr.us-east-1.amazonaws.com/jenkins_dv:latest
```

You can run the Jenkins master as an Amazon ECS service using a Docker compose file, which starts the Jenkins master and the Jenkins data volume containers. When running as an Amazon ECS service the Jenkins master will restart if its container fails or stops.

Create a Docker compose file with the entries below:

```
jenkins_master:
 image: jenkins_master
 cpu_shares: 100
 mem_limit: 2000M ports:
 ports:
  - "8080:8080",
   "50000:50000"
 volumes_from: jenkins_dv
jenkins_dv:
 image: jenkins_dv
 cpu_shares: 100
 mem_limit: 500M
```

Use the Amazon ECS command line to start the Jenkins master container as a service:

```
# Create an ECS service from Jenkins compose file
ecs-cli compose --file compose file service up
```

The Jenkins master should also be configured with AWS credentials that give the Jenkins master the appropriate privileges to register task definitions, run tasks, and stop tasks in the Amazon ECS cluster.

Setting Up Jenkins Builders

To set up the Docker image used by the Jenkins master to start Amazon ECS Jenkins worker tasks, use the Jenkins continuous integration (CI) official image for the Docker-based JNLP worker as the base image in the Dockerfile. Then add any additional tools needed to support the build process into the Dockerfile. Store the custom Jenkins worker image created with the Dockerfile in Amazon ECR.

CloudBees also provides a Docker image for JNLP worker nodes that includes the tools to build Java applications that can be used as the image to run Amazon ECS workers.

Use the custom Jenkins worker image to configure the worker template in the Jenkins master. The worker template specifies the label used to refer to the worker, the Docker image used to create the worker, and the resources (such as CPU and memory) needed by the worker Amazon ECS task.

When creating a job in the Jenkins master you can specify which type of Amazon ECS worker you want for the job. Just specify the label used in the

Amazon ECS worker template configuration in the **Restrict where this Project can run** field.

Security Considerations

Security Groups

The security considerations when running the Jenkins master and worker nodes in Amazon ECS are similar to the traditional deployment of Jenkins. The security group associated with the container instances in the cluster should only allow TCP traffic t0 8080 and only allow access to port 5000 on container instances from other container instances in the clusters.

Using AWS CodeBuild with Jenkins

AWS CodeBuild is a fully-managed build service in the cloud. AWS CodeBuild compiles your source code, runs unit tests, and produces artifacts that are ready to deploy.

AWS CodeBuild

AWS CodeBuild eliminates the need for you to provision, manage, and scale your own build servers. It provides prepackaged build environments for the most popular programming languages and build tools such as Apache Maven, Gradle, and more. You can also fully customize build environments in AWS CodeBuild to use your own build tools. AWS CodeBuild scales automatically to meet peak build requests, and you pay only for the build time you consume.

AWS CodeBuild provides the following benefits:

- **Fully Managed** – AWS CodeBuild eliminates the need for you to set up, patch, update, and manage your own build servers. There is no software to install or manage.
- **Secure** – With AWS CodeBuild, your build artifacts are encrypted with customer-specific keys that are managed by the AWS Key Management Service (KMS). CodeBuild is integrated with AWS Identity and Access Management (IAM), so you can assign user-specific permissions to your build projects.
- **Continuous Scaling** – AWS CodeBuild scales automatically to meet your build volume. It immediately processes each build you submit and can run separate builds concurrently, which means your builds are not left waiting in a queue.
- **Extensible** – You can bring your own build tools and programming runtimes to use with AWS CodeBuild by creating customized build environments in addition to the prepackaged build tools and runtimes supported by CodeBuild.
- **Enables Continuous Integration and Delivery (CI/CD)** – AWS CodeBuild belongs to a family of AWS Code services, which you can use to create complete, automated software release workflows for CI/CD. You can also integrate CodeBuild into your existing CI/CD workflow. For example, you can use CodeBuild as a worker node for your existing Jenkins server setup for distributed builds.
- **Pay as You Go** – With AWS CodeBuild, you are charged based on the number of minutes it takes to complete your build. This means you no longer have to worry about paying for idle build server capacity.

Jenkins and AWS CodeBuild Integration

At a functional level, Jenkins has two components: a powerful scheduler that allows the creation and execution of complex jobs, and a build platform, namely, the worker nodes. Jenkins provides most of its value in job orchestration. The Jenkins build workers are undifferentiated. This means that they can be replaced by other systems that are more efficient or more effective, and, as we've seen in previous sections, can live and die with each build with no impact to the business. For that reason, they are a perfect candidate for being offloaded to a managed service like AWS CodeBuild.

The Jenkins AWS CodeBuild Plugin, which we show you how to use later in this whitepaper, allows for integrating AWS CodeBuild within Jenkins jobs. This means that build jobs are sent to the AWS CodeBuild service instead of to Jenkins worker nodes (which are not needed anymore). This eliminates the need for provisioning and managing the worker nodes. The following diagram shows a typical setup for this.

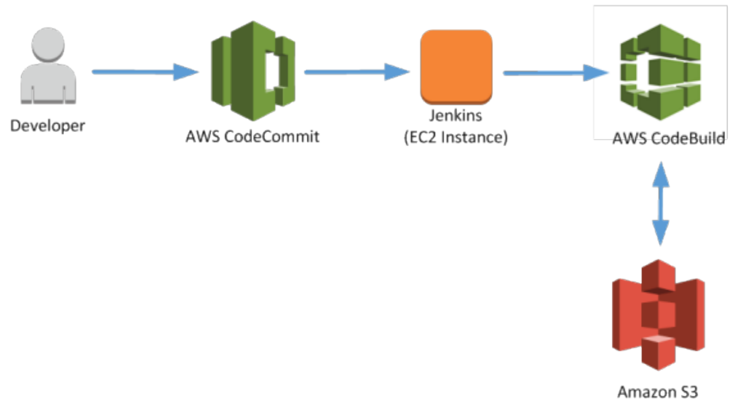


Figure: Jenkins integration with AWS CodeBuild

To set up the integration of the AWS CodeBuild service and the Jenkins server, perform the following steps:

1. Build the Jenkins AWS CodeBuild plugin (aws-codebuild.hpi). This plugin will be used to run the AWS CodeBuild projects from the Jenkins server.
2. Install the Jenkins AWS CodeBuild plugin onto the Jenkins server.
3. The plugin will need an IAM identity in order to run the AWS CodeBuild projects from the Jenkins server. To do this, create an AWS IAM user to be used by the plugin. The access/secret key information associated with the new IAM user will be used to configure the build step for the Jenkins project in the later step.
4. The plugin will also require certain IAM permissions in order to perform the AWS CodeBuild actions, access resources in Amazon S3, and retrieve Amazon CloudWatch logs. To do this, create an AWS IAM policy with the required permissions and attach the policy to the AWS IAM user created in the previous step.
5. Create an AWS CodeBuild project for the actual build step. To do this, create and configure an AWS CodeBuild project in the selected AWS Region and ensure it can build the target project successfully directly inside of the AWS CodeBuild. This project will be invoked by the plugin on the Jenkins server.
6. To configure the plugin to invoke the AWS CodeBuild project from the Jenkins server, create a freestyle project in Jenkins server. Next, add a **Build Step** and choose **Run build on AWS Codebuild**. Configure the build step with information such as the AWS Region and the project name of the AWS CodeBuild project created previously, as well as the IAM user access/secret key details created previously.

After you have completed these steps, you will be ready to run the build command in Jenkins to send the build job to the AWS CodeBuild service. For detailed instruction, see the awslabs-codebuild-jenkins-plugin on GitHub to run a build using AWS CodeBuild.

Jenkins Integration

One of the reasons that Jenkins is widely popular is because it integrates with many third-party tools, including AWS products. Each of the five solutions that follow provides a compelling use case for Jenkins.

AWS Integration

The plugin ecosystem for Jenkins offers options for integration with these AWS services:

- Amazon EC2
- Amazon ECR
- Amazon Simple Notification Service (SNS)
- Amazon ECS
- Amazon S3
- AWS CloudFormation
- AWS CodeDeploy
- AWS CodePipeline
- AWS CodeCommit (Note that the AWS DevOps Blog offers additional insights on how to integrate AWS CodeCommit with Jenkins.)
- AWS Device Farm
- AWS Elastic Beanstalk

The solutions that follow walk you through how to use these services together to create useful patterns.

Other Notable Plugins

The following plugins are widely recognized by Jenkins communities. The solutions focus on AWS integration. If you don't find what you are looking for, you can see the full list of Jenkins plugins on the official site.

- SSH Slaves - Manage workers through SSH
- JUnit - Graphical representation of your JUnit tests
- Maven Project Plugin - Build Maven projects
- Mailer - Email notifications for builds
- Monitoring - Chart server information (resource consumption, server activity, etc.)
- Nested View Plugin - Groups jobs into multiple levels instead of a single big list of tabs
- Build Monitor Plugin - Visual view of your builds status
- Dashboard View - Portal-like view for Jenkins
- Green Balls - Changes Hudson to use green balls instead of blue for successful builds

Solution 1: Vanilla CI/CD Pipeline

Many AWS customers host their code, builds, and applications on AWS, and use AWS CodePipeline for orchestration.

Goal

In this solution we discuss how the pipeline works, relying on Jenkins for deployment. This typical workflow allows product teams to release to their customers quickly and collect feedback in a timely manner.

Services Used

- Amazon EC2
- AWS CodeCommit
- AWS CodeDeploy
- AWS CodePipeline

Architecture Diagram

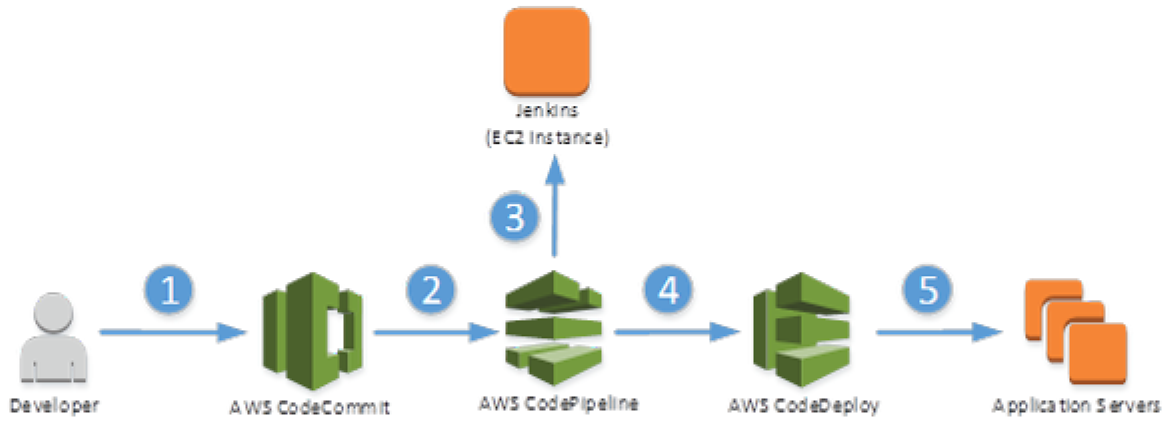


Figure: A simple workflow using AWS services and Jenkins

Explanation

1. Developer commits code to AWS CodeCommit using a standard git commit command.
2. AWS CodePipeline detects that new code has been pushed to AWS CodeCommit and triggers the pipeline.
3. AWS CodePipeline invokes Jenkins to build the application.
4. Upon a successful build, AWS CodePipeline triggers deployment on AWS CodeDeploy.
5. AWS CodeDeploy deploys the application onto AWS application servers.

This example does not make a distinction between quality assurance (QA), staging, and production environments. This would involve additional steps (either automated or manual) for more complex DevOps workflows than the one we discuss here.

Note that AWS CodePipeline supports other repositories like GitHub and other deployment options like AWS Elastic Beanstalk and AWS OpsWorks. Find out how to build a pipeline in the AWS CodePipeline documentation.

Solution 2: Container Pipeline

To simplify and speed up the delivery process, a number of customers building micro service-based architectures are adopting containers as their unit of deployment.

Goal

This solution allows the developer to have his or her changes packaged into a Docker image and spun up as a container in the different environments associated with the stages in the delivery pipeline.

Services Used

- AWS CodeCommit
- Amazon EC2
- Amazon ECR
- Amazon ECS

Architecture Diagram

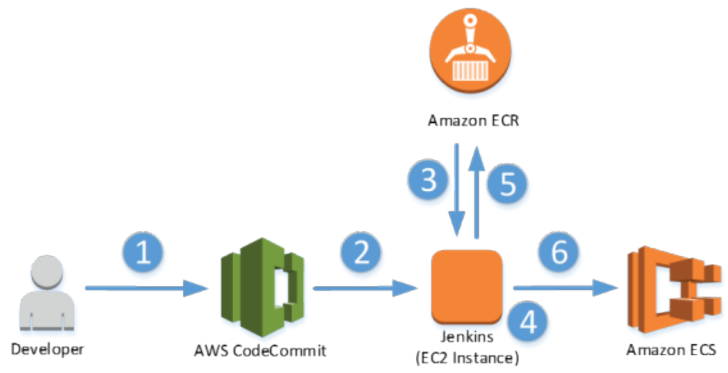


Figure: Using AWS services and Jenkins to deploy a container

Explanation

1. Developer commits code using a standard git push command.
2. Jenkins picks up that new code has been pushed to AWS CodeCommit.
3. Jenkins pulls a Docker image from Amazon ECR.
4. Jenkins rebuilds the Docker image incorporating the developer's changes.
5. Jenkins pushes updated the Docker image to Amazon ECR.
6. Jenkins starts the task/service using the updated image in an Amazon ECS cluster.

Find out how to deploy such a pipeline on our Application Management Blog.

Solution 3: Mobile Application

AWS Device Farm allows AWS customers to test their iOS or Android mobile applications on a fleet of physical devices. It allows them to detect bugs and optimize for performance. Jenkins integrates well with AWS Device Farm.

Goal

In this scenario, a developer and a tester (who could be the same person) upload work to AWS CodeCommit. Jenkins picks up this new code and pushes it to AWS Device Farm for testing.

Services Used

- Amazon EC2
- AWS CodeCommit
- AWS Device Farm

Architecture Diagram

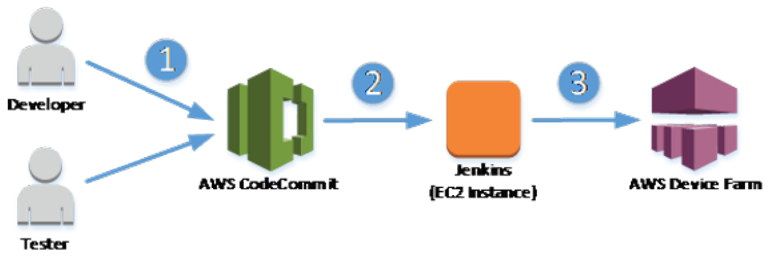


Figure: Using AWS services and Jenkins to test a mobile application

Explanation

1. A developer and a tester use Eclipse to generate a Maven build, which gets pushed to AWS CodeCommit.
2. Jenkins picks up the new code that has been pushed to AWS CodeCommit and pulls the application.
3. Jenkins pushes the application and the tests to AWS Device Farm for testing across a pool of physical devices.

Find out how to deploy such a pipeline on the AWS Mobile Development Blog.

Solution 4: Serverless Code Management

You can use AWS Lambda, an Amazon S3 bucket, Git, and Jenkins for a serverless code management solution.

Goal

In this solution, an AWS Lambda function monitors an Amazon S3 bucket for new PDF objects. Developers check the function code into a Git repository, and Jenkins hooks are used to build and test the latest code changes. The latest successful builds are pushed back into AWS Lambda.

Services Used

- AWS CodeCommit
- Amazon EC2 (for the Jenkins deployment)
- Amazon S3
- AWS Lambda
- AWS DynamoDB

Architecture Diagram

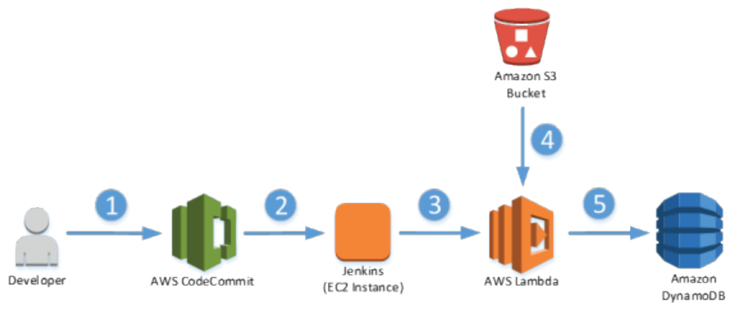


Figure: Using AWS and Jenkins for serverless code management

Explanation

1. Developers work independently on code and store their work in a Git repository like AWS CodeCommit.
2. Git hooks, like post-commit or post-receive, are used to notify Jenkins that a build is required.
3. Jenkins runs the build, executing any available unit tests, and invokes a post-build step that uses the AWS CLI to upload the latest code package to AWS.
4. The AWS Lambda function picks up on new objects being added to the Amazon S3 bucket.
5. Objects are processed with the latest code, and results are written into Amazon DynamoDB.

Find out how to deploy such a pipeline on our AWS Compute Blog.

Solution 5: Security Automation Framework

When moving to a CI/CD model, application teams aim to rapidly release and update code. However, they often become blocked by manual security processes that are required when performing deployments.

Goal

This solution allows security teams to automate their security processes and integrate them into the deployment pipeline, thus allowing application teams to scale their pace of deployment without compromising the overall security of the application.

For this solution we have two examples:

Example 1: A developer commits a code change to an existing AWS CloudFormation template, which is then combined with another AWS CloudFormation template from the security teams.

Example 2: A developer commits an update to his or her application code, including AWS CodeDeploy's AppSpec.yml. Here, Jenkins merges requirements from the security team into that AppSpec.yml.

Services Used

- Amazon EC2 (for the Jenkins deployment)
- Amazon S3
- AWS CloudFormation
- AWS CodeDeploy
- AWS CodeCommit

Architecture Diagram – Example 1

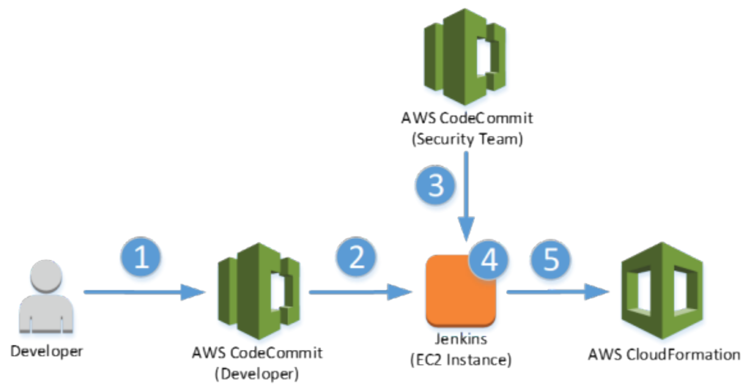


Figure: Using AWS services and Jenkins to automate a security framework

Explanation – Example 1

1. Developer commits his or her CloudFormation JSON file to the application AWS CodeCommit repository.
2. Jenkins picks up the change in AWS CodeCommit on its next polling interval and kicks off new build process.
3. Jenkins pulls down the security controls for CloudFormation from the security teams' AWS CodeCommit repository (e.g., allowed IP address ranges, allowed AMIs, etc.).
4. Jenkins executes the security checks against the developer CloudFormation template. The job only proceeds if it doesn't encounter violations. (If it does encounter a violation, it fails and can alert the security team.)
5. Upon success, Jenkins issues an update stack call to CloudFormation using the validated template.

Architecture Diagram – Example 2

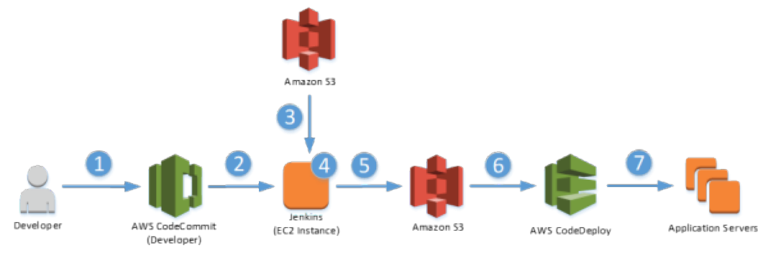


Figure: Using AWS services and Jenkins to automate a security framework

Explanation – Example 2

1. Developer commits an application code update to the application AWS CodeCommit repository.
2. Jenkins picks up the code change in AWS CodeCommit on its next polling interval and kicks off new build process.
3. Jenkins pulls the security portion of the AppSpec.yml file for AWS CodeDeploy from the security teams' Amazon S3 bucket.
4. Jenkins merges this with the developer/application teams' AppSpec.yml file.
5. Jenkins zips this all up and sends it to an Amazon S3 bucket.
6. Jenkins kicks off an AWS CodeDeploy API call. EC2 instances pull the zip file from the Amazon S3 bucket and deploy.
7. AWS CodeDeploy returns success only if all checks from the security teams' AppSpec.yml succeed.

Conclusion

Faster software development has become a competitive advantage for companies. The automation of software development processes facilitates speed and consistency.

Jenkins is the leading automation product. We see many of our most successful AWS customers implementing it. This whitepaper walks you through using AWS services with Jenkins and covers some of the common customer scenarios for AWS integration with Jenkins.

Document Details

Contributors

The following individuals and organizations contributed to this document:

- Nicolas Vautier, Solutions Architect, Amazon Web Services
- Jeff Nunn, Solutions Architect, Amazon Web Services Special thanks to the following individuals for their contributions:
- Chuck Meyer, Solutions Architect, Amazon Web Services
- Michael Capicotto, Solutions Architect, Amazon Web Services
- David Ping, Solutions Architect, Amazon Web Services
- Chris Munns, DevOps Business Development, Amazon Web Services

Document History

Date	**Description**
May 2017	Inclusion of AWS CodeBuild
September 2016	First publication

Resources

When you want to dive deeper into the topics of this whitepaper, the following resources are great starting points:

- Continuous delivery on AWS: http://aws.amazon.com/devops/continuous-delivery/
- Continuous integration on AWS: http://aws.amazon.com/devops/continuous-integration/
- DevOps and AWS: http://aws.amazon.com/devops/
- Jenkins on the AWS Marketplace: *https://aws.amazon.com/marketplace/search/results/ref=dtl_navgno_s earch_box?page=1&searchTerms=jenkins*

- AWS Architecture Center
- AWS Whitepapers
- AWS Answers
- AWS Documentation

AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the *AWS General Reference*.