Cloud Development Reference Model

April 2014

1.0 **Why Use the Cloud?**

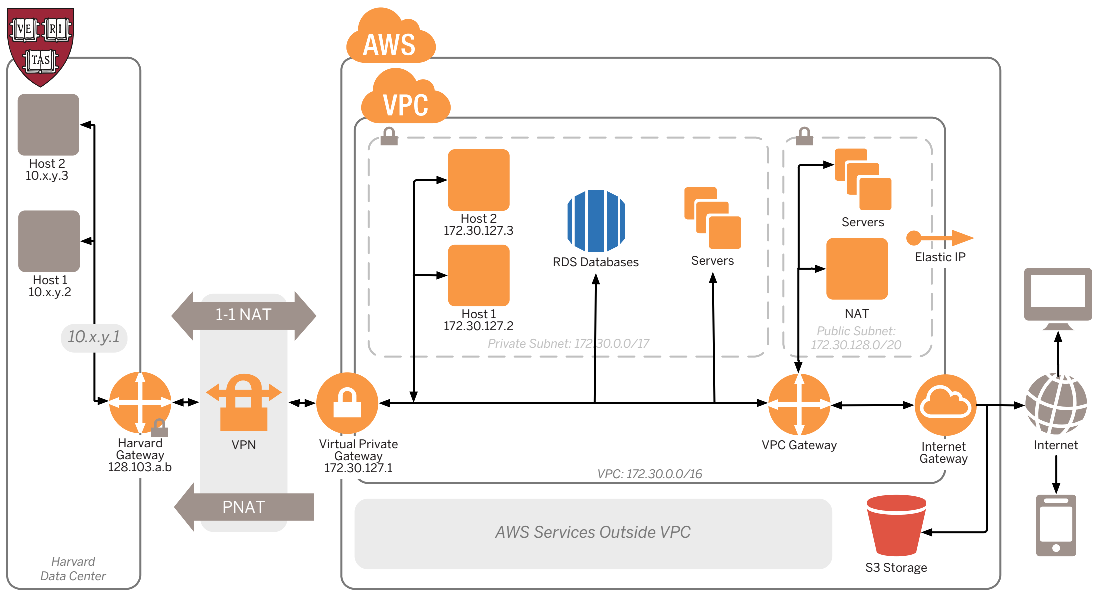
Traditionally, deployments require applications to be bound to a particular infrastructure. This results in low utilization, diminished efficiency, and inflexibility. The cloud enables applications to be dynamically deployed onto the appropriate infrastructure at runtime, an elastic aspect that allows applications to scale and grow on demand without needing traditional patches or upgrades.

This IAM Development Reference Model describes the key infrastructure components required for Cloud adoption.

2.0 **Network View: Infrastructure Hosted on Amazon Web Services (AWS)**

In this model, the bulk of traffic — including all user traffic — passes across the internet gateway to AWS' publicly routable network space. (Some privileged traffic is allowed over a back-end VPN connection to Harvard network space.) Most of this traffic is PAT'd through a single IP address on Harvard network space, but direct one-to-one NATing is also available if needed.

This model has only a loose dependence on internal Harvard network services, but access is still required. Ideally, these dependencies are not required for normal runtime functioning of hosted applications, but instead are limited to management traffic, secure data movement, and other cron or batch jobs.



Note: Single AZ depicted.

Addressing

VPC: 172.30.[0.1-255.255], mask 255.255.0.0
Private Subnet: 172.30.[0.0-127.255], mask 255.255.128.0
Public Subnet: 172.30.[128.0-143.255], mask 255.255.240.0

Private Route Table

Local: 172.30.0.0/16
VGW: 10.0.0.0/8
NAT: 0.0.0.0/0

Public Route Table

Local: 172.30.0.0/16
IGW: 0.0.0.0/0

Diagram 2.0.1: Network View: Infrastructure Hosted on AWS

2.1 **Multiple Availability Zones**

For production applications, infrastructure components and instances should span more than one availability zone (AZ) in an AWS region — at least two. This protects against loss at the data center level.

For each AZ, you will need to duplicate subnets, instance pools, NAT instances, and storage. Objects such as load balancers and databases will need to be spanned across AZs, and auto-scaling groups should be defined to assure that instances are distributed across AZs, usually with at least one instance available in each AZ.

2.2 **ELB Subnets**

AWS recommends as best practice placing elastic load balancers (ELBs) into AWS's own subnets, separate from instances. This simplifies configuration, since ELBs host no data or application logic.

2.3 **Data Subnets**

For ease of auditing and securing sensitive data, it may make sense to place databases and instances hosting sensitive data at rest on dedicated subnets. This allows for explicit ACLs and rules limiting access to these target services.

2.4 **Public Versus Private Subnets**

In general, placing hosts on the public subnet is a simpler, cleaner configuration, since each host is automatically NAT'd to the Internet; this simplifies patching, upgrading, and access to services and data stored outside the VPC (such as S3 buckets, SQS queues, etc.). Using security groups on these hosts in the public subnet provides strong isolation from publicly routed traffic while also allowing for traffic egress. The use of the private subnet should, in general, be limited to guests that require more complete isolation from the Internet. In order to access data and services outside the VPC, NAT instances are required in each AZ, and currently are single points of failure. Because of this, access to the Internet and publicly routed AWS networks would not be in the critical path for production functioning of these hosts.

2.5 **Use of NAT and Proxy Instances**

For instances hosted on private subnets, the general model is to use NAT hosts and routes to provide access to services outside the VPC, which then are single points of failure. While you can place a NAT instance into an auto-scaling group that assures that it is restarted if terminated, the use of a single guest limits bandwidth.

2.6 **Planning VPC Network Space for Growth**

Even if you are using a VPC model that is isolated from the data center network, if there is a chance you may want to link multiple VPCs in the future (using a VPN connection) and allow routing between the two VPCs, you should plan for this now in how you choose your network space. One example is to use 172.30.0.0/16 for one VPC, and 172.29.0.0/16 for the next. This allows for more than 30 VPCs to be interconnected without IP overlap. Similarly, for VPCs isolated from the Harvard network, network spaces such as 10.X. 0.0/16 allow for upwards of 256 non-overlapping VPC networks.

3.0 **Web Application Network View**

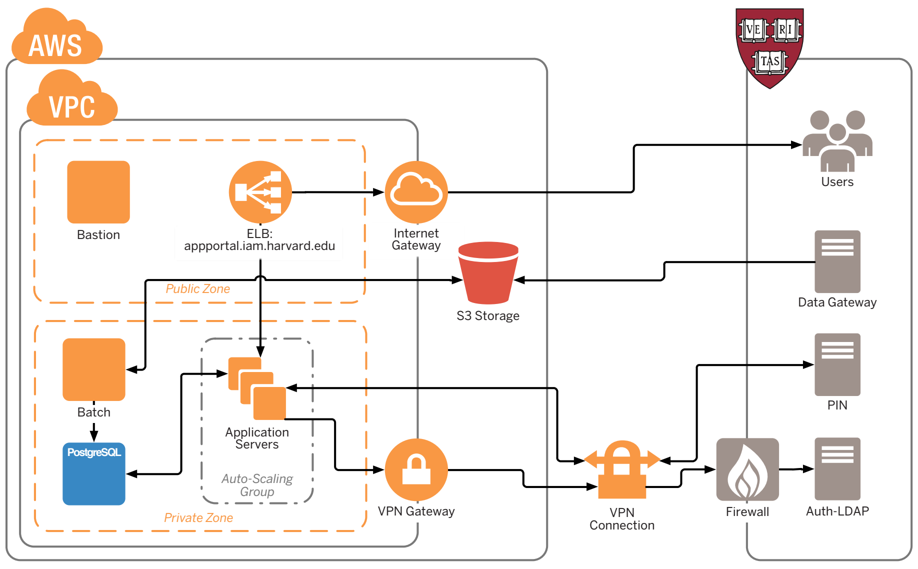


Diagram 3.0.1: Web Application Network View

3.1 **Key Points**

- The Web application interacts with end users via an elastic load balancer hosted in the public zone and accessed via Internet gateway
- The ELB is set with an SSL certificate stored within AWS IAM
- All data resides in a PostgreSQL RDS instance, hosted in the private zone of the VPC, which has a connection to the HUIT data center via a VPN connection
- Initial data loads and incremental updates are supplied from internal HUIT resources via the S3 bucket
- The batch server creates Initial data structures and populates the database with initial data and incremental updates from the HUIT data center
- The application interacts with the database and private HUIT data center resources, and resides in an auto-scaling group in order to provide scalability and redundancy across two AZs
- The application is configured and deployed by a puppet module

3.2 **Major Configuration Components**

- Web server: Apache 2.4
- mod_auth_cas apache module for CAS based authentication
- Shibboleth service provider for SAML2-based authentication
- The Web application resides in a Java container hosted by Apache Tomcat 7 (requires jdk7)

4.0 **Continuous Delivery**

Continuous delivery enables the automation of the entire software delivery system — from development build and deployment to test and release to production — through a series of steps known as the delivery pipeline.The diagram below describes the continuous delivery workflow for creating an AWS AMI from the development environment and using it in upper environments.

The Jenkins continuous integration (CI) server pulls the code changes in the source repository and does the deployment in the development environment. When development reaches a milestone (a stable state), the CI server creates an AWS AMI based on a predefined trigger. This AMI is used in all upper environments with environment-specific configuration.

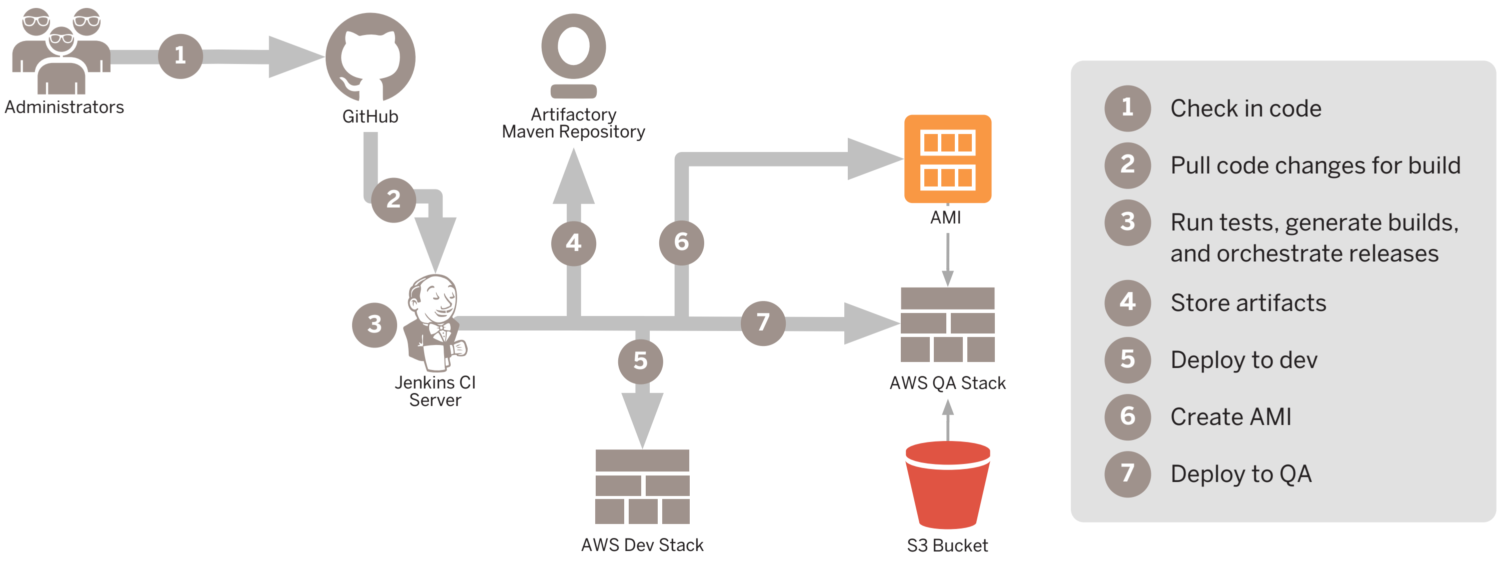


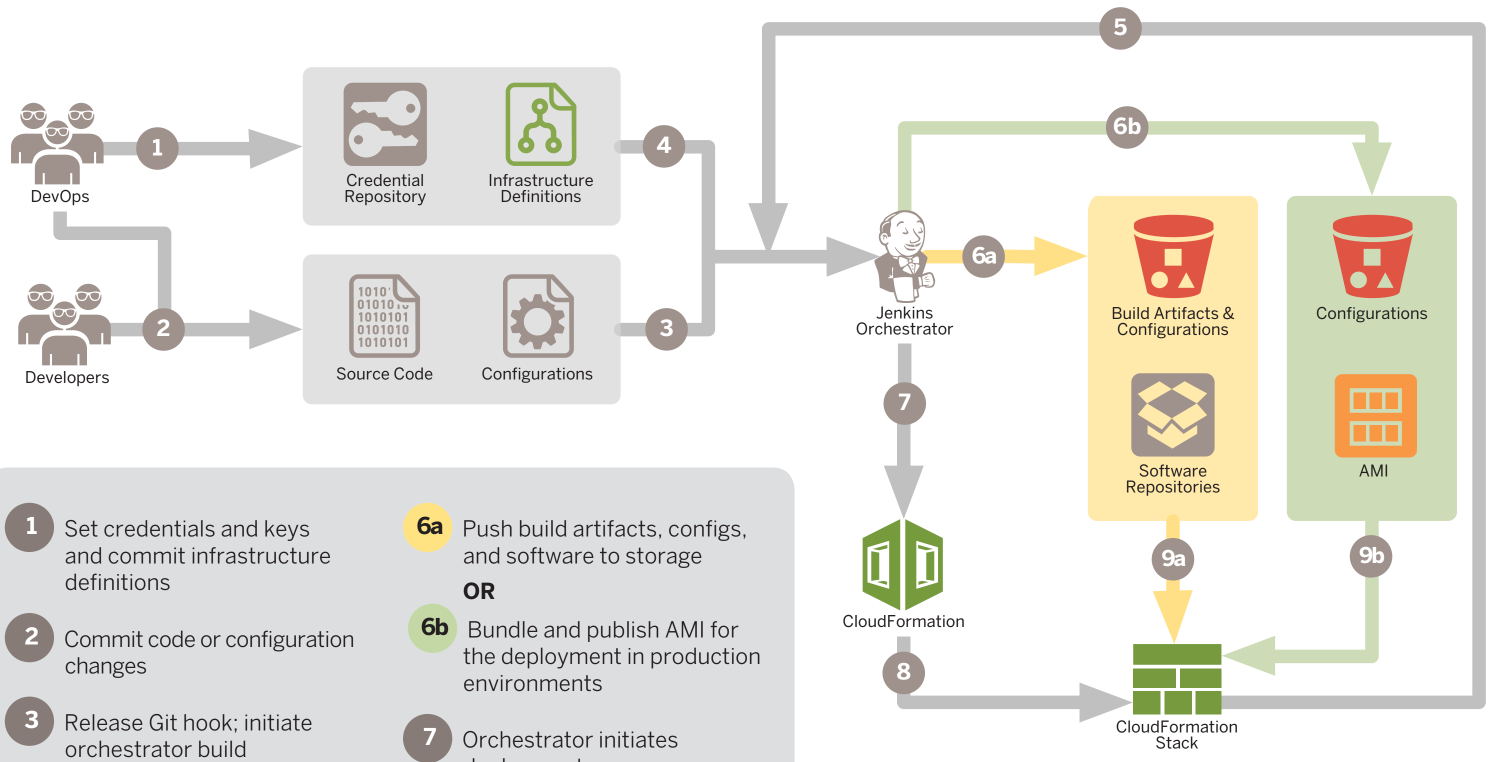
Diagram 4.0.1: Continuous Delivery Workflow

5.0 **Web Application Deployment Workflow**

Deployment activity follows the completed development process, including all operations necessary to prepare a system for assembly and transfer to the stack running on AWS. CloudFormation templates determine the resources required to operate the stack. The Jenkins orchestrator collects information from resources that are already running on AWS, as well as external resources running at Harvard, for carrying out subsequent activities in the deployment process.

When an application has gone through all development and quality assurance steps, the AMI that is provisioned by the development process can be released to a stage environment for final validation. The AMI is then released to a production AWS account and ready for final testing; if critical bugs are found, we return to development release workflow iterations.

In the diagram below, differences in workflow between development and production environments are represented in yellow and green, respectively.



1. Set credentials and keys and commit infrastructure definitions
2. Commit code or configuration changes
3. Release Git hook; initiate orchestrator build
4. Set credentials and infrastructure definitions via CloudFormation templates
5. Retrieve info about existing CloudFormation resources to configure infrastructure

6a Push build artifacts, configs, and software to storage

OR

6b Bundle and publish AMI for the deployment in production environments

7 Orchestrator initiates deployment

8 CloudFormation creates or updates stack

9a Pull artifacts, software, and configurations

OR

9b Bundle image with deployed software

Diagram 5.0.1: Web Application Deployment Workflow

5.1 Key Points: Deployment Flow in Development Environment

- Deployment is initiated by changes in Git source control repositories and processed by the orchestrator, which represents a set of tools (discussed below)
- Generated artifacts and configurations are stored on S3 as a WAR archive and set of configuration templates
- Configurations are generated based on external sources including Credential Store (to store passwords and certificates), a git repository with configurations for puppet modules, the current state of the existing environment (captured from outputs of running CloudFormation stacks), and tagged resources of the given environment
- The application stack is deployed by a CloudFormation template based on artifacts and configurations captured earlier in the process
- An Amazon Machine Image (AMI) is created from application server built as part of the application stack after all tests are executed and passed against the given stack
- The released AMI is shared with all IAM AWS accounts and ready for the release process represented on the diagram above

5.2 Key Points: Deployment Flow in Production Environment

- In the process of the AMI release, the orchestrator collects configurations from the credentials repository and runs CloudFormation stack for the application that is to be deployed
- A release is done by updating auto-scaling groups with the new AMI and trigger update of the given CloudFormation stack