UP AND RUNNING

A guide to building a cloud-based environment that is secure, scalable, and suited to modern development needs

Element 84

Table of Contents

Background

In technology-driven businesses, the challenges of limited budgets, growing data volume, and broadening developer needs are increasingly driving corporations and government agencies of all types into the public cloud. The availability of managed disk space, computational resources, and an ever-growing set of features make the cloud far more attractive than hiring, building, and maintaining significant infrastructure and the personnel required to maintain them.

What's Hard About a Cloud-Based Environment?

As of Q1 2018, 33 percent of the cloud market has utilized the Amazon Web Services (AWS) platform (see https://www.cnbc.com/2018/04/27/microsoft-gains-cloud-market-share-in-q1-but-aws-still-dominates.html). As a result, many companies are now in the process of setting up cloud-based environments. Element 84 is often asked, "How can we build our own AWS environment?" There are a few other variants to this question as well:

- "Our developers want to deploy their applications to AWS. How can we allow that and remain secure? Is AWS secure?"
- "We need a research and development environment (or labs environment, or staging environment) with controlled spending. How do we do that?"
- "I thought the whole point of the cloud was to not need a bunch of administrators. How can we support an AWS environment without help?"

All of these questions boil down to, essentially, "How do we use the cloud intelligently?" The answer we provide is nearly always the same: a resounding "It depends…" In this whitepaper, we will outline some common themes and best practices for integrating with AWS.

We'll begin by making clear that moving to the cloud is far more than a technical commitment. There are significant changes in thinking that must occur. Understanding how responsibility changes from an on-premises environment to a cloud one is key in structuring your environment. While many organizations start with "shadow IT"–more on that soon–it should only be a step along the path toward a well-maintained, documented, and centrally managed cloud environment.

Then there’s billing. If any one topic creates confusion in the cloud, it’s billing. This is not so much an issue of “how much do things cost” as it is one of “how do we keep that new guy from spinning up 200 instances of his testing environment and blowing the whole month’s budget?” A solid account structure can handle this issue, as well as lay the foundation for a solid networking approach.

And networking is key; the default setup from AWS will work, but it isn’t nearly as resilient and secure as the approach we lay out. The AWS tools are all at your disposal, but you’ll need to use them properly. Here’s where bastion hosts, public and private subnets, VPCs, and security all enter the picture.

Finally, you’ll want to take everything you’ve done and automate it. Anything you have to do manually once, you should try and avoid having to do manually twice. CloudFormation is key here, as is configuration as code.

Shifting to a Cloud Paradigm

The first step in putting together a useful, effective, cost-controlled cloud environment is not a technical step at all. Before working in the cloud, you need to grasp some key concepts related to cloud environments. These concepts don’t map directly to on-premises environments, and can make the difference between a clunky and difficult-to-maintain cloud environment versus one that elevates your organization.

Who is Responsible?

Many of these paradigm-shifting issues arise when routine on-premises questions are asked of cloud deployments. For instance, take the reasonable question, “Who stood up this server?” In an on-premises world, this is equivalent to asking, “Who was responsible for plugging in, cabling, networking, provisioning, etc., this physical server?” The answer is often a single name (“Bob did!”) or perhaps a team or shift (“The overnight SA group did!”). However, in a cloud world, this question and answer is more nuanced. A DevOps engineer or Systems Administrator might have created a new instance on AWS, but it might have just as likely been a predetermined scaleup that happened because resources reached a certain threshold, and new instances were automatically started up as part of an auto-scaling process. In that case, who did stand up the server? The engineer who setup the auto-scaling group? AWS itself? It’s not as clear an answer.

This is where the cloud paradigm shift occurs: responsibility is not always as clear cut in a cloud-based world. This becomes even more apparent when you begin to take advantage of the very services that make cloud attractive: elasticity (the ability to expand resources on demand), auto-scaling (the ability to add resources on demand), and auto-provisioning (setting up templates that run in certain situations, often without human manual intervention). But it is not enough to simply wave a hand and say, “That’s not how it works in the cloud.” Traceability, access, resource management, and security all are first-class citizens in a cloud world; they just aren’t organized and managed in the same way that they are in an on-premises world.

It often is the job of whomever is championing or leading a cloud transition to understand these questions in the cloud context, and communicate the right answers. That means that you may need to be fluent in the language of security, networking, and compliance–beyond your own field of DevOps or application programming–to effect a healthy move to an AWS cloud environment.

Shadow IT is Not a Viable Solution

Shadow IT is a term used to represent employees in an organization that circumvent normal process and procedure, instead using an unsanctioned cloud environment–AWS or otherwise. While this problem is most common in large organizations where pockets of cloud deployments are most easily hidden and confined to the fringes, this can be a problem for CTOs and organizations of any size. Some recent surveys suggest that as many as 80 percent of employees of typical large organizations are using or have used a non-approved software-as-a-service (SaaS) solution (see the Rackspace paper referenced in Appendix 1, “Managing the Transition to IT as a Service Broker”).

This shadow IT–sometimes also called “Bring Your Own Cloud”–is the opposite of the cloud paradigm shift we recommend. Internal, non-sanctioned cloud deployments fragment IT resources, as something will inevitably go wrong and affect the larger organization. Further, the ease with which uncontrolled cloud environments allow practically anyone to spin up their own environment means that often important network-level decisions are made without enough input from the resources that run the corporate network. The result? Long and costly migrations after the initial “migration to AWS” project when the rogue environment’s hosted applications inevitably must be moved to the “real” networks for official launches.

> While it is often the case that these rogue environments are pockets of unsanctioned accounts in AWS, they sometimes crop up as non-cloud environments with the same set of problems. A server in a closet that everyone is afraid to touch can have the same detrimental effects to an organization’s consistency and security as an unknown cloud account.

If you have these environments floating around, consider them positive in that they reflect an interest in moving toward the cloud, as well as a desire by your employees to move their projects forward. They may even be creating an experience for your team that is valuable when brought to bear on an official environment. Just make sure that as you do create your official environment, you work to shut down or absorb the unofficial ones. Also, keep in mind what drove these users outside in the first place: a desire for flexibility. If your official cloud implementation isn't as flexible, or as productive as the one they just left–chances are you haven't seen the last of shadow IT.

The Cloud Paradigm is Agile but Still Careful

In cases where shadow IT is prevalent, or in smaller organizations where moving to the cloud is seen as "a way to not keep hiring up to support our needs," there is a common misunderstanding: the cloud is a "quick" way to get things moving. While the cloud paradigm provides an empirically faster path to deploying applications and supporting enterprises, it is still an environment that requires care and caution.

These two words–care and caution–have a bad connotation in most organizations, especially among developers. They are often translated to, "everything will move at a snail's pace." But this is not the case; care and caution implies **purpose** and **involving the right people at the right time**. The network engineers and system administrators and security personnel that set up an on-premises network need to be involved in your cloud buildout, but the processes and guidelines they used in that setup are going to have to evolve.

Adopting a cloud-centric approach in your organization, then, should reflect a focus on the cloud as a mechanism not for hosting applications, but as an environment that promotes efficiency, resource management, and evolution as an organization's needs exchange. This is exactly where we have found AWS to excel–when used as a well-understood IT framework (rather than shadow IT) with a cloud-aware philosophy rather than a dated on-premises one.

Accounts are Your AWS Building Blocks

When you begin to build your your cloud environment, top-tier concerns should be to address responsibility, security, and what may be the most-often cited reason for moving to the cloud, cost. Beginning with an organizational approach to your cloud environment focused on accounts is ideal… and gets at the need to provide a responsibility model that supports budgets and controlled spending.

Create a Master Billing Account

Your first step is to setup a master billing account, as shown in Figure 1.

FIGURE 1 - You need a top-level account when you create your AWS account structure to be a parent to all billing sub-accounts

Accounts in AWS provide organizational structure for billing and security. You can further fine-tune your cloud management using AWS Organizations (https://aws.amazon.com/organizations/). This allows you to segregate infrastructure into a master billing account and individually scoped sub-accounts.

You'll use the master billing account to pay all the consolidated bills across sub-accounts, as well as create logins–and as little of anything else as possible. In other words, this is a mostly cost-focused construct, and can be locked down to personnel who need to deal with and track these costs. Your larger IT organization only needs minimal access to login in and manage their credentials, and that's a good thing for the responsibility and traceability your cloud setup should preserve.

> AWS does provide another option often used for billing: tags (https://aws.amazon.com/answers/account-management/aws-tagging-strategies/). Using tags, you can associate certain instances or services with particular tags, and then organize and track billing based on those tags. While this was popular in AWS's earlier days, though, this is no longer our recommended approach. Tags are prone to errors in entry (misspellings and capitalization being the most common offenders) and not all AWS services support tags, nor do all support tags in the same manner. The result is often a billing report based on tags that is only a partial total, as compared with actual overall bills. We have found that using sub-accounts is a much better and more effective approach.

Create Segmented Sub-Accounts

Next, you need to segment your costs. This segmentation can be as simple or as complex as you like, and each segment will become a sub-account underneath the master billing account. Here are several examples of segmentation:

- Grouping production accounts together, and then grouping non-production accounts together
- Segmenting by department, for example, Platform, Data Access, Web, Back-End, etc.
- Segmenting by application, with each application having its own sub-account
- A combination of all of these. For example, application 1 from the Platform team has a production, staging, and development sub-account, application 2 has the same, etc.

Ultimately, you will likely have multiple scoped sub-accounts, and those should be created underneath the Master Billing Account, as shown in Figure 2.

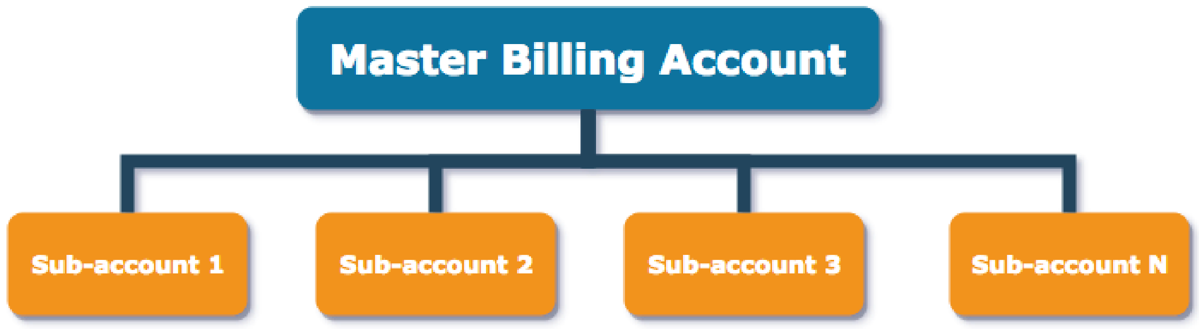


FIGURE 2 - The master billing account has underneath it your various sub-accounts, each of which should be mapped to a segment that has business value to your organization.

As the number of sub-accounts grows, you will run into a new set of problems: sub-accounts can only be one level deep for billing purposes. This means if you segment in a way that is extremely granular, you could end up with 50, 75, or more sub-accounts underneath the master billing account. Keep this in mind as you're balancing the need for billing granularity with management overhead. There's no wrong way to divide it up, but there are trade-offs to those decisions.

> This step can be quite tedious if you have a lot of segments for which you're creating sub-accounts. Still, that tedium is worth the work, as your efforts at this early stage have a huge effect on accurate billing and reporting.

Manage Access to AWS Services with Sub-Accounts

Recall that one of the key challenges and paradigm shifts associated with a cloud environment is responsibility. Closely related to that is access; it is often better to ensure that only the right people can take an action as the first step toward assigning and determining responsibility. This is another area where a Master Billing Account and intelligently designed sub-accounts are a huge first step toward a manageable AWS environment.

> Terminology around IAM is a little confusing. Permissions are generally written at the level of a"policy." and then that policy is applied to a user, a user group, or a role. Where things start to get tricky is across accounts or sub-accounts. At first login, policies are granted by user and user group, but once you cross accounts, they are granted by roles. (If you needed to read that sentence more than once, you're not alone!) For ease of conversation here, we'll use "roles" as a general umbrella of user groups and roles.

Every account in AWS has access to what AWS calls their IAM service, the Identity and Access Management service. This service is the key to allowing and disallowing access to AWS services–almost all of which have an associated cost–and then aligning that access with your sub-accounts. An IAM role can allow or deny access to a set of services. Figure 3 is a very simplistic example; this IAM policy allows everything to whomever its assigned.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

FIGURE 3 - IAM provide a traceable and simple mechanism for allowing (and disallowing) access to resources to accounts and their users.

For many new organizations, though, IAM is used on a per-user basis. You should immediately see how unwieldy this becomes: with even 20 users, the permutation of service/access level/user means you'll quickly have hundreds if not thousands of IAM policies to keep up with… and eventually, audit.

Thankfully, AWS has an answer for this built into IAM for organizations. Sets of permissions in IAM are organized into policies, and policies are applied to users, user groups, or roles. This is how most organizations think about permissions anyway: a specific set of permissions is created for a group of people–like application developers–and then users are assigned to that group. The result is permission management is handled at a logical role level, rather than an individual user level.

Figure 4 shows the definition of a set of permissions via IAM, and that set of permissions being assigned to a specific role.

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "arn:aws:iam::<subacctNum>:role/<roleName>"
    }
}
```

FIGURE 4 - IAM is best used to assign permissions to a group or role, rather than a specific user. This creates a more resilient permission structure than direct assignment to users, in the general case.

Roles can also be assumed across accounts, meaning that if we create a single user in the master account, that user can use the assume role system to perform their job duties across all the sub-accounts in the organization, based on either their user or user group policies, or both. If you created a sub-account through the AWS Organizations UI or the API, there will be an admin role created for you in the sub-account (Figure 3) that has a trusted entity of the master account number, and a policy created in the master account that allows you to assume that role (Figure 4). Existing accounts can also be invited to an organization, and set up the same way as a sub-account - a very useful way to pull shadow IT accounts back into the organization without requiring huge migration efforts.

> A note on AWS SSO - AWS now provides a Single Sign on Service (SSO) that wraps some of the steps provided in this section into a managed service. For some organizations, this will end up being a better choice than the steps we provide above. However, like any choice - there are trade offs. For us - when we were making this decision - the CLI access procedures for SSO were not yet as user friendly as the methods above - and with a large number of power users, programmatic access needed to be solid. Of course, we expect this to change as AWS improves the SSO service based on customer feedback.

Create and Secure Your Network

With a healthy master billing account and sub-accounts in place, you need to next develop your network model. It may seem tempting to drill further down into your billing at this stage, setting up account limits and cost caps. However, your network is going to have a significant impact on how those costs accrue, and how you control them. You will not be able to construct sensible limits until you understand usage patterns. Getting the network right is the first step in understanding these patterns.

When you create an AWS account or sub-account, AWS creates a default virtual private cloud (VPC) in all regions. VPCs are customer-managed virtual networks; you can check out the AWS documentation at https://aws.amazon.com/vpc/. You'll have a VPC in every region you have access to - for most users, that's 18 VPCs at current count. That's quite convenient–you get these VPCs for free–but it's going to give you trouble down the road. These VPCs all have the same network space allocated to them, and that means that routing between VPCs is problematic, requiring network address translation (NAT) of some sort, or even more complex network wrangling.

For this reason, we recommend not using this default VPC setup in any of your regions, or at all. Instead, building your own VPCs and adopting a hub-and-spoke network model is a much better option.

DMZ Subnets and Hub-and-Spoke Networks Increase Security

In the default AWS configuration, each VPC uses the network CIDR of 172.31.0.0/16. That means that two VPCs in two regions actually have the same address range, making routing between those two VPCs complicated, at best, and in some cases, outright impossible. Obviously, that's not ideal.

Additionally, the default model does not use a DMZ model for the subnets that are created for you. This is done for a reason: it ensures that the VPC will not incur any charges for a NAT gateway. However, this cost savings comes at a different type of price: you have a much wider threat profile as a result.

> A DMZ stands for *demilitarized zone*, and is a special network configuration that improves security by segregating computers (or, in the cloud, applications and services) on each side of a partition, usually implemented by a firewall and/or network routing. Most often this separation is between Internet accessible services and private ones.

A hub-and-spoke network model for VPC-to-VPC communications, as well as the creation of DMZ subnets within those VPCs, provides a couple of key benefits over the default VPC model:

- Key resources and administrative access can be routed via secure VPN and peering links, with traffic coming from known private networks.
- Creation of DMZ subnets keeps your publicly-accessible threat profile to a minimum, as well as providing an extra layer of security against misconfiguration breaches.
- Using separate, routable VPCs also makes troubleshooting and triage easier, as each VPC has its own discrete network block.

The hub and spoke network model allows you to secure both your AWS resources against a breach in your local network, and protect your local network from a compromise in your

AWS infrastructure. With a little planning on your part, you can make the robust routing capabilities of your VPCs work for you, rather than creating the problems discussed above.

For the most part, routing tables in VPCs look very similar to routing tables in a traditional data center. However, there is one key difference that you can use to your advantage: transient routing is never allowed. This is important because it prevents one compromised machine from infecting your entire network.

Consider VPCs A, B, and C, where traffic from B to A is routable and B to C is routable. A packet originating in VPC A, then, intended for VPC C, is never passed through VPC B–even if there are valid network routes that existing from A to C through B. In a practical application, this means that a worst case compromise of a resource in VPC A cannot impact VPC C without a second intermediary compromise of a resource in VPC B (Figure 5).

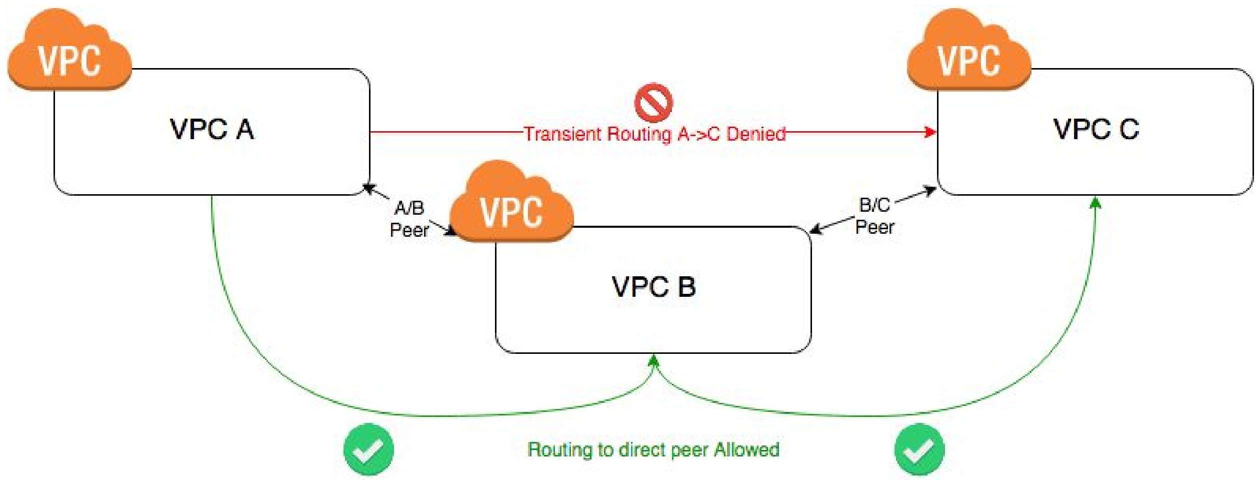


FIGURE 5 - Transient routing from VPC A -> C is not allowed, despite sharing a peer. You must hop A->B->C.

Organize Individual VPCs Around a Bastion Sub-Account

At Element 84, we use this restriction against transient routing to secure access to resources at a network layer as part of our layered security model. Figure 6 shows a basic hub-and-spoke layout that builds on the master account and sub-account concept discussed in the section above, [Accounts are Your AWS Building Blocks](#).

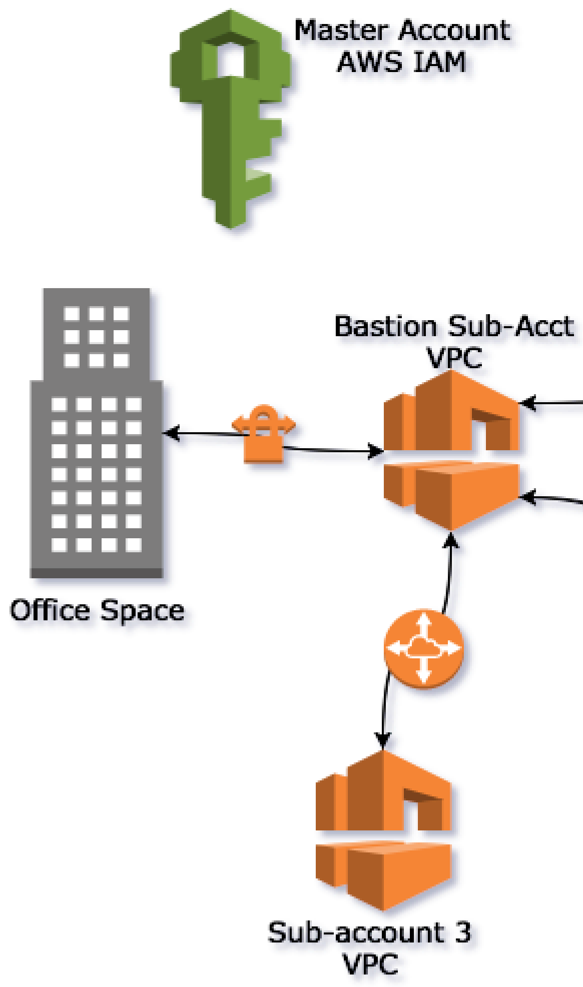


FIGURE 6 - A hub-and-spoke network model segregates resources and only allows access through a central bastion account.

First, notice that there are no VPCs in the master account. The master account is really nothing more than a logical construct in this diagram, and that's intentional. This is the account that "owns" all the rolled up billing information, and can handle top-level authentication. It should not, though, have networking attached.

Rather, the VPCs all exist at the sub-account level. This provides all the advantages and protections of VPCs for each account–which typically represents a specific application and environment. That means you're separating these applications and environments from each other, which is a primary feature of VPCs.

Then, the individual sub-account VPCs are connected by a special type of VPC-to-VPC link called a VPC peering connection. The link to the local office in the diagram uses an AWS VPN connection service. And in each case where a VPC is connected to another VPC or to a local office, there's something in the middle: a bastion VPC (also sometimes referred to as a "transit VPC"). A bastion VPC is simply a variation on a DMZ: a place used for access that also isolates unrelated VPCs. In this network design, the bastion is functioning as the DMZ, a "neutral zone" through which all traffic flows. While that does mean you'll have to manage what can be pretty complex routing tables in the bastion VPC, all those tables are in a single central location, which is extremely helpful as a network gets large.

In a good network design, the bastion further ensures a separation of security concerns. Remember that restriction against transient routing? The bastion host, acting as a sort of hub between the various individual VPCs, further enforces that restriction. It ensures that routing is centrally controlled and traffic is well-segregated between sub-accounts.

> While transient routing is an actual AWS restriction, using a bastion host is not a hard requirement. You actually can peer VPCs directly to each other; however, you lose the clarity of security boundaries that are shown in the figure above. Unless you have a very specific reason to directly peer VPCs, you should route through a bastion host whenever possible.

Build a Layered Security Model Into Your VPCs

So far, the focus has been on the overall network, and how VPCs are positioned relative to each other. Now, let's take a closer look at how each individual VPC is configured. Figure 7 shows a typical setup where a VPC has multiple subnets, both public and private.

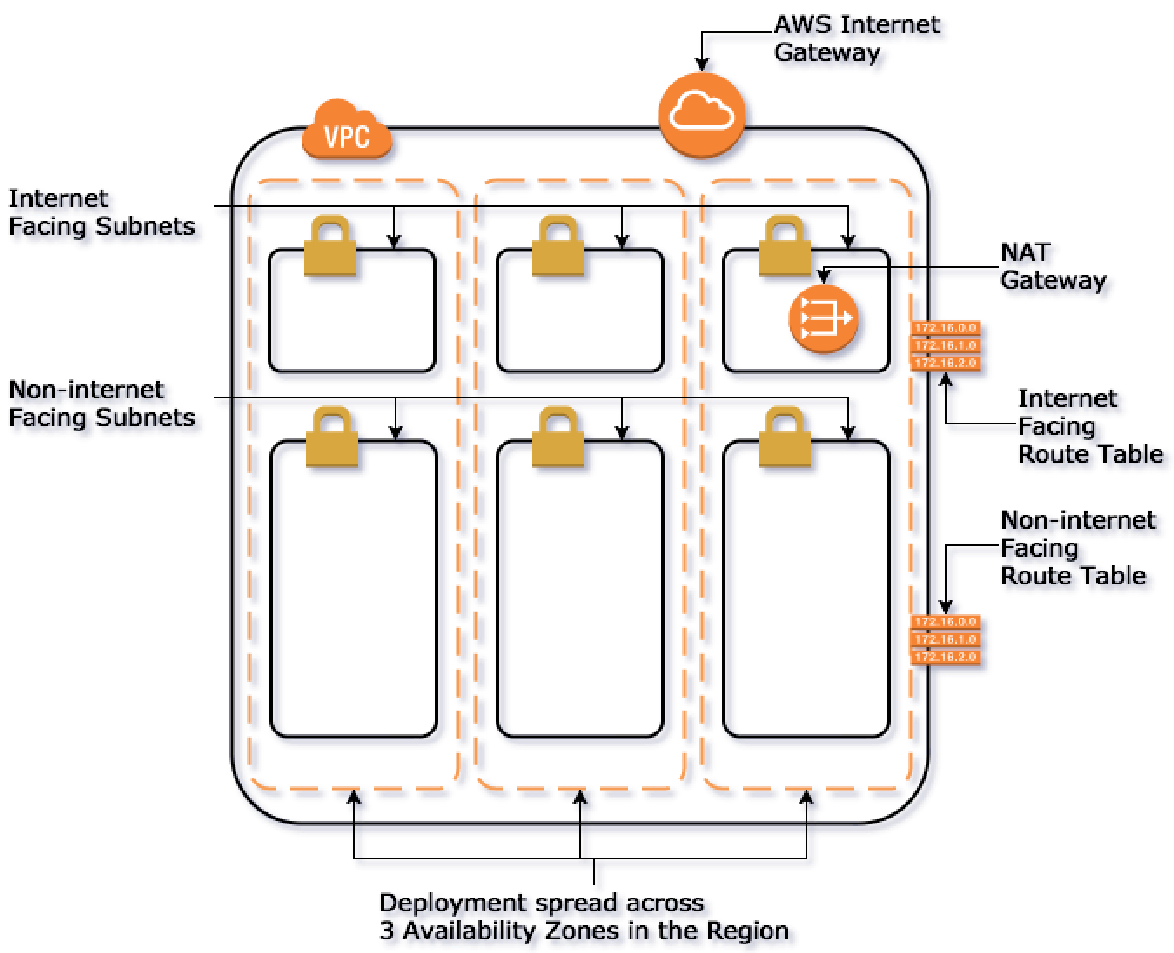

AWS Internet Gateway
VPC
Internet Facing Subnets
NAT Gateway
172.16.0.0
172.16.1.0
172.16.2.0
Internet Facing Route Table
Non-internet Facing Subnets
Non-internet Facing Route Table
172.16.0.0
172.16.1.0
172.16.2.0
Deployment spread across 3 Availability Zones in the Region

FIGURE 4 - A highly available AWS network with public and private subnets, where all subnets require Internet access

This setup has multiple private subnets that only are routable to the Internet by way of a NAT gateway. As many of your lambdas and EC2 instances should be placed in these private subnets, you gain another layer of security by not directly exposing any of them to direct Internet access.

> This approach to VPC design isn't just something we require in our own offices; it's actually required for some important certifications, like PCI (see https://www.pcisecuritystandards.org/ for more).

The advantage of a layered security model is easily demonstrated with a simple example. In Figure 8, we lay out two identical deployments, each with the common misconfiguration of entering the wrong address block for the private 172.16.0.0/12 network as defined in RFC 1918. Instead, the CIDR of 172.0.0.0/8 is used.

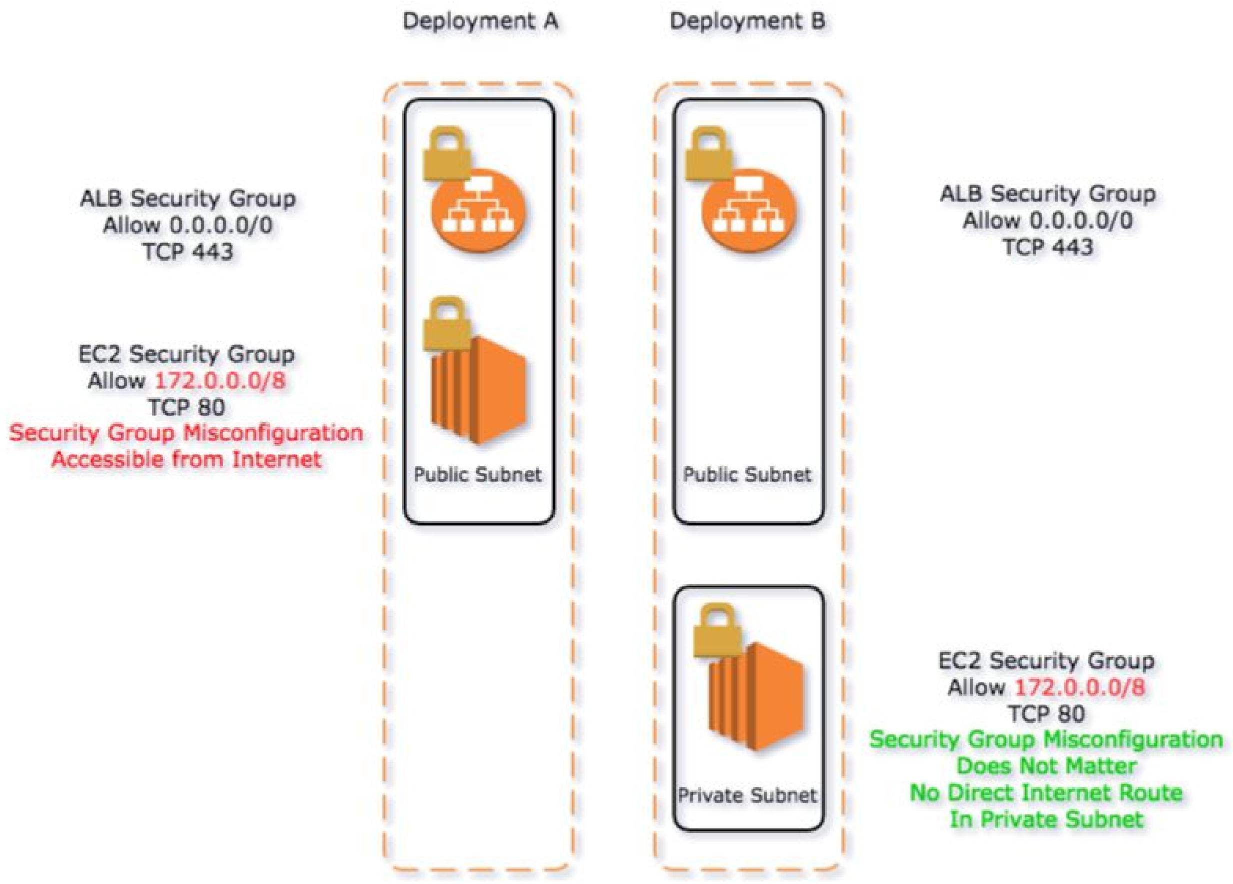


FIGURE 8 - Layered network security model with private subnets can prevent issues due to misconfiguration.

In both deployments (Figure 8), SSL traffic from the Internet is terminated on the Application Load Balancer (ALB) and then passed to the backend servers, where port 80 is open to the overly broad network. In Deployment A, which uses a layout similar to the default VPC setup in AWS, this misconfiguration will lead to port 80 on the backend servers being exposed to any public IP that's in the 172.0.0.0/8 network, which is not part of the RFC 1918 reserved private IP space. To get a real handle on what that means, that's almost *16 million* internet IP addresses. In Deployment B, because the backend EC2 is in a private subnet, there is no added exposure from this misconfiguration because the layer of security above the security group catches it.

While it's easy to think, "I simply won't make that misconfiguration," it's also just as plausible that someone else will... and why err on any side but the secure, cautionary one?

This is why using private subnets and a layered network security model within each sub-account is so critical.

Reduce, Restrict, and Remove Host-Level Access

The last layer of security to discuss, at least for the scope of this paper, is that of host level access. This is going to be controversial and likely unpopular with many, but reducing–and ultimately removing–host-level access to your system for all but your most privileged users is an essential step in securing your network.

One major advantage of cloud computing is the elasticity of the environment. Hosts can be added to handle an increase in traffic, and those hosts can just as quickly be destroyed when no longer needed. Now imagine that each of those hosts can be accessed by administrators, operators, developers... anyone who has general access to your environment. And with each access, changes can be made, creating one-off hosts that are similar to the other hosts, but aren't quite exactly the same. These changes, though, last only as long as that particular host is available. When it goes away, the process often begins again, as changes need to be made again, and the cycle continues.

In a development environment, this level of access often makes sense, and might even be required. Developers need to debug failed deployments or misbehaving code, and it's often far more efficient to hop onto a host and see what's going on. However, this practice wreaks havoc in testing and production environments.

> There are good arguments to be made for slowly weaning off most console access even in lower environments. At a minimum, this process can help facilitate building tools that make console access unnecessary in testing and production environments. This is beyond the scope of this paper, and locking down hosts at the production levels would be a fantastic and significant step toward a secure AWS environment.

Begin this process by asking whether your staff needs console access at all. If the answer is "No," then congratulations! You have the easiest and most secure access plan: just remove access. Make sure there are no firewall rules allowing console access, and turn off the services that provide that access if possible. For example, on a linux EC2 instance you would disable ssh on the server and configure the security group to not allow traffic over port 22.

If there are legitimate reasons that some of your staff needs host-level access, that's ok, too; there's just a little more work involved. The goal is to fluidly add/remove new hosts and onboard/offboard end-users. We recommend using solving this problem with a Directory Service, such as Active Directory, OpenLDAP, or even AWS-hosted Directory Service (https://aws.amazon.com/directoryservice/).

> We're big proponents of Open Source Software at Element 84, so if you don't have a directory service, we'd recommend FreeIPA (https://www.freeipa.org/). FreeIPA is the upstream project of RedHat's IPA server, and you can easily drop in the RedHat version if you need enterprise support. We're going to talk about the steps needed in general terms though, so any directory service should work.

Now you need to attach your hosts to the directory server and use the directory server for authentication and to restrict access. If you're familiar with a data center server, you have these basic steps in a server's lifecycle:

1. Server is created and brought up on the network
2. Authentication against the Directory Server is configured
3. Some piece or pieces of software are installed and configured
4. That software does some (hopefully) useful work for a time
5. Server is decommissioned and removed from the Directory Server if needed

The lifecycle of an application server in AWS is very similar; the only notable difference is the scale of the creation and decommissioning. For the purposes of providing host access, the key steps here are 2 and 5, both obviously the steps related to a directory server. In each of these cases, you need to automate the process of authenticating against the directory server (step 2) and then removal (step 5). Automation here is crucial, because you could have hundreds (and in highly volatile enterprise solutions, thousands) of hosts cycling up and down in a single day.

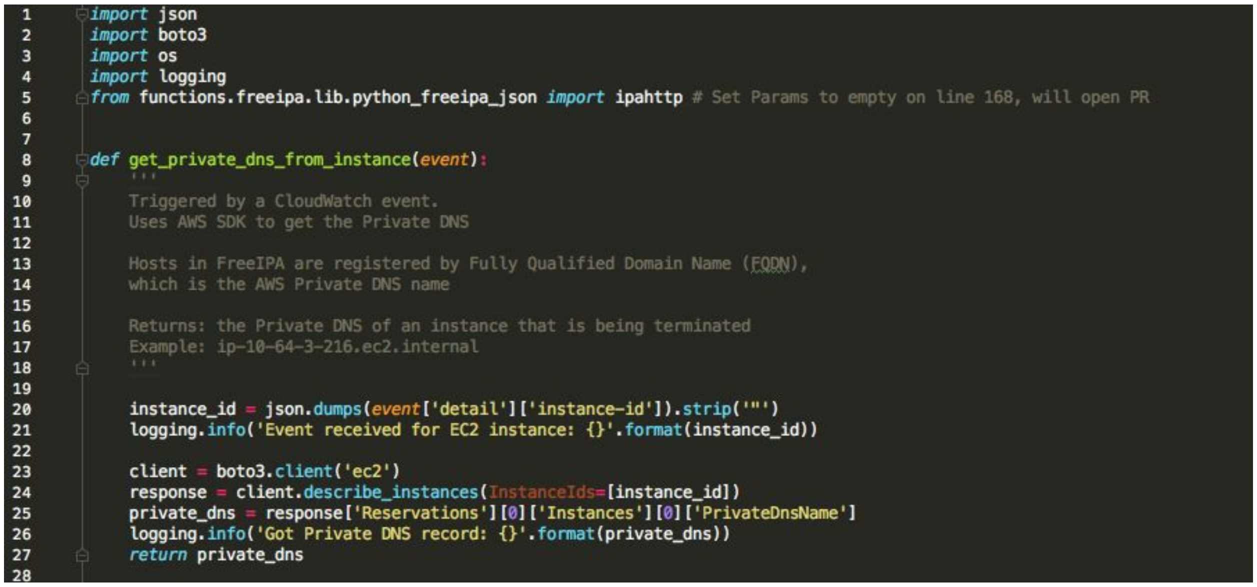
Joining a Directory Server

First you need to set your host to authenticate or join your directory server. This is a straightforward process, but will vary some with different directory servers and tools. For example, with FreeIPA, there's a client install utility that configures the System Security Service Daemon (or SSSD) on a AWS Linux-based host. This configuration causes the host to join to the IPA server, making it a managed resource. This managed resource can then be configured and interrogated in a central location–along with all the other hosts similarly configured. You'll find this process changes a bit from directory server to directory server, but generally has the same result: central management and identity control. This takes care of authentication of the host.

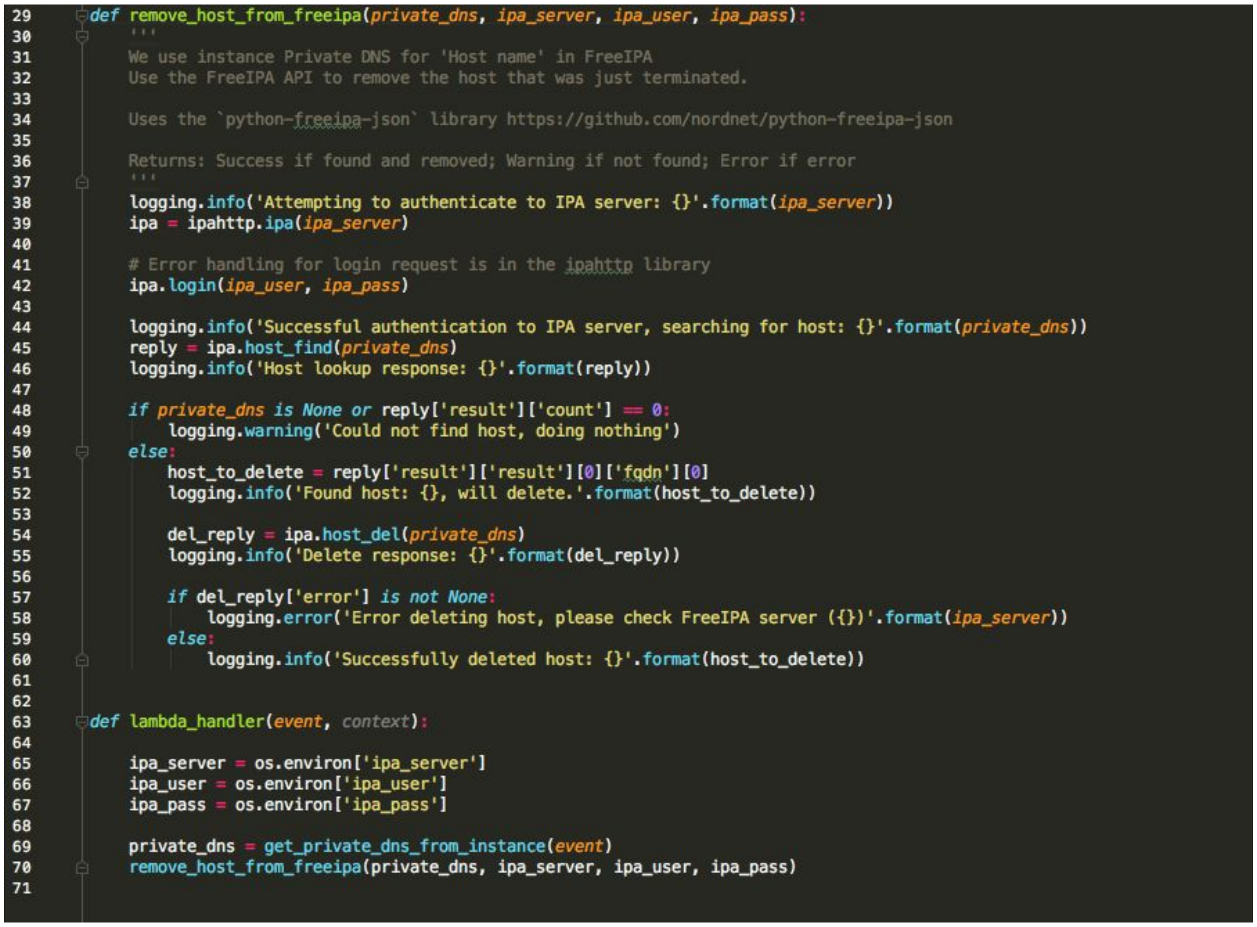
Leaving a Directory Server

Decommissioning is not quite as simple. Each entry in your directory server for a host has to be removed when that host is terminated. There are no native directory server tools to address this; you will have to rely on AWS tools. One option leverages a combination of CloudWatch (https://aws.amazon.com/cloudwatch/), the AWS monitoring system, and Lambda (https://aws.amazon.com/lambda/), AWS's serverless code platform, outlined below.

CloudWatch gives you a set of events that trigger at certain points in a host's lifecycle–like when the host is getting ready to decommission. You can attach a predefined bit of Lambda code to that event, which is exactly what is needed here. In this case, some simple Python will do the trick; keep in mind that this code is FreeIPA specific (Figure 9).



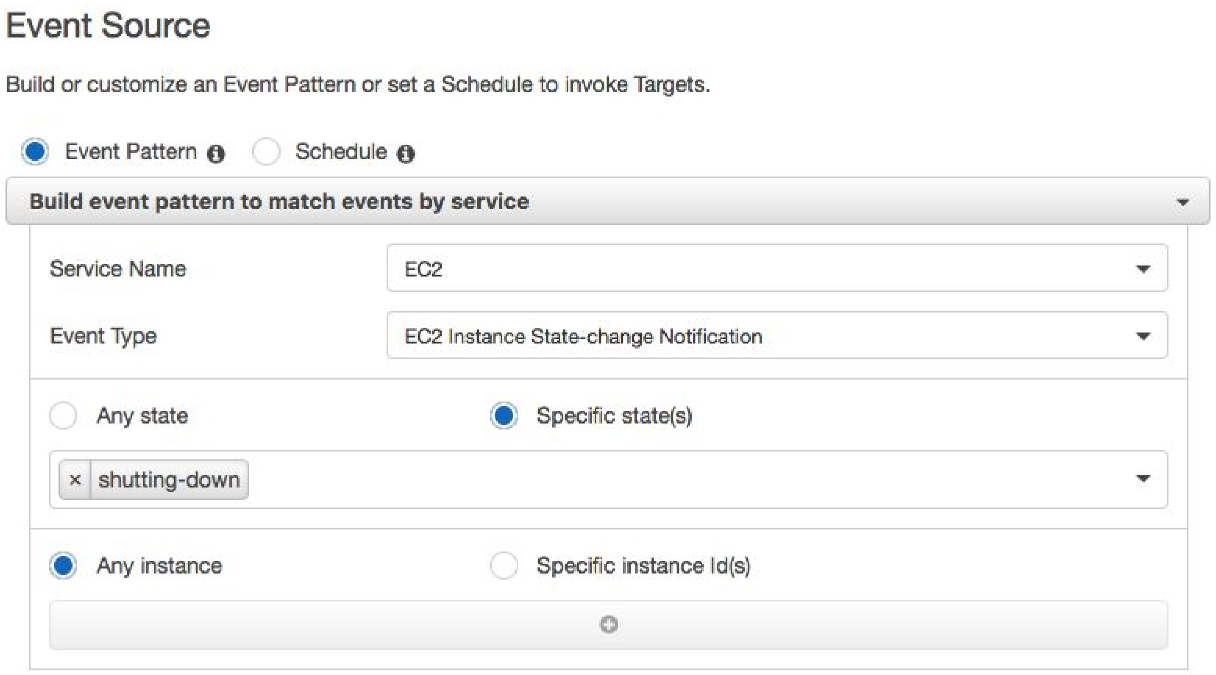
```python
import json
import boto3
import os
import logging
from functions.freeipa.lib.python_freeipa_json import ipahttp # Set Params to empty on line 168, will open PR


def get_private_dns_from_instance(event):
    '''
    Triggered by a CloudWatch event.
    Uses AWS SDK to get the Private DNS

    Hosts in FreeIPA are registered by Fully Qualified Domain Name (FQDN),
    which is the AWS Private DNS name

    Returns: the Private DNS of an instance that is being terminated
    Example: ip-10-64-3-216.ec2.internal
    '''

    instance_id = json.dumps(event['detail']['instance-id']).strip('"')
    logging.info('Event received for EC2 instance: {}'.format(instance_id))

    client = boto3.client('ec2')
    response = client.describe_instances(InstanceIds=[instance_id])
    private_dns = response['Reservations'][0]['Instances'][0]['PrivateDnsName']
    logging.info('Got Private DNS record: {}'.format(private_dns))
    return private_dns

```



```python
def remove_host_from_freeipa(private_dns, ipa_server, ipa_user, ipa_pass):
    '''
    We use instance Private DNS for 'Host name' in FreeIPA
    Use the FreeIPA API to remove the host that was just terminated.

    Uses the `python-freeipa-json` library https://github.com/nordnet/python-freeipa-json

    Returns: Success if found and removed; Warning if not found; Error if error
    '''
    logging.info('Attempting to authenticate to IPA server: {}'.format(ipa_server))
    ipa = ipahttp.ipa(ipa_server)

    # Error handling for login request is in the ipahttp library
    ipa.login(ipa_user, ipa_pass)

    logging.info('Successful authentication to IPA server, searching for host: {}'.format(private_dns))
    reply = ipa.host_find(private_dns)
    logging.info('Host lookup response: {}'.format(reply))

    if private_dns is None or reply['result']['count'] == 0:
        logging.warning('Could not find host, doing nothing')
    else:
        host_to_delete = reply['result']['result'][0]['fqdn'][0]
        logging.info('Found host: {}, will delete.'.format(host_to_delete))

        del_reply = ipa.host_del(private_dns)
        logging.info('Delete response: {}'.format(del_reply))

        if del_reply['error'] is not None:
            logging.error('Error deleting host, please check FreeIPA server ({})'.format(ipa_server))
        else:
            logging.info('Successfully deleted host: {}'.format(host_to_delete))


def lambda_handler(event, context):

    ipa_server = os.environ['ipa_server']
    ipa_user = os.environ['ipa_user']
    ipa_pass = os.environ['ipa_pass']

    private_dns = get_private_dns_from_instance(event)
    remove_host_from_freeipa(private_dns, ipa_server, ipa_user, ipa_pass)

```

FIGURE 9 - Lambda code from Element 84 open source project at the time this document was produced.

You can download this code for yourself at https://github.com/Element84/cloudwatch-deregister-ec2-hosts. We've also included in the repo an IAM policy to attach to the Lambda function in order to work with CloudWatch.


Event Source

Build or customize an Event Pattern or set a Schedule to invoke Targets.

(● Event Pattern) ○ Schedule

Build event pattern to match events by service

Service Name: EC2

Event Type: EC2 Instance State-change Notification

○ Any state (● Specific state(s))

× shutting-down

(● Any instance) ○ Specific instance Id(s)

FIGURE 10 - Setting up a CloudWatch event to trigger the Lambda on EC2 instance termination. We use "shutting-down" not "terminating" because the network configuration is still available in the event context.

With this Lambda code written, you just need to associate it with a CloudWatch event when a server is decommissioned (see Figure 10). You can do this using the AWS console too, either on the CloudWatch event page or the Lambda configuration page.

Build Scalability Into Your Network

We often hear from smaller organizations new to the cloud that they want to make sure they don't overcommit and overspend in the cloud, but they need to be able to handle explosive growth. These organizations usually have smaller user bases that spike up at certain times. On the other hand, larger organizations with millions of customers may need large banks of hundreds or even thousands of application instances to serve their user base. In both of these instances–and all the use cases that fall in between these extremes–the issue at hand is scalability.

Scalability Begins with Principled Design

While scalability surfaces most often as a technical issue, it's really about design and careful thinking. There are three core principles that should guide your design:

- Prefer redundancy over single instances
- Prefer automation over manual process
- Prefer repeatability over one-off actions

In each of these principles, you can really read “prefer” as “do everything in your power, and then some to ensure…” In other words, it’s worth making different design choices–and incurring additional work–to implement designs that embody these principles. That’s because initial work with these principles in mind will almost always result in a more than comparable reduction in work in ongoing operations and maintenance.

By preferring redundancy over single instances, you ensure that you always have two or more instances, or containers, or service endpoints running. You avoid the dreaded case of “my server went down” because there’s never just one server (or container, or database, or API endpoint).

By preferring automation over manual process, you ensure that anything that needs to be done can be done quickly and in a consistent manner. Instead of a 4 page email with lots of numbered steps, you have a single script, checked into version control, that is the source of truth when it comes to configuring a new ElasticSearch instance or database schema. You’re never at the mercy of “that engineer that worked here last year, right before Dave came on. Remember him? What was his name again?”

By preferring repeatability over one-off actions, you isolate and remove edge cases. Your bank of instances are all identical, and can be treated uniformly. You’re no longer managing a system of snowflakes that are each fed and cared for individually. This is a case where uniqueness is not a desired quality!

Put these three principles together and you have a centrally managed, easily controlled uniform environment that can be managed and scaled through scripts, not a fleet of system administrators.

Enhance Your Network with Redundancy

So how do these principles look when applied in practice? Refer back to Figure 7, a VPC framework that’s the basis of the network recommendations in this paper. Let’s enhance that infrastructure by using ALBs, application load balancers (see https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html for more on ALBs).

> For simplicity's sake, this example doesn't show redundancy at the NAT gateway level. That's easy to add, though. If you've been using automation, it's often just a single parameter change in your VPC's CloudFormation.

Figure 11 is an example of taking an application deployment and adding ALBs and a few additional AWS services to implement scalable design principles.

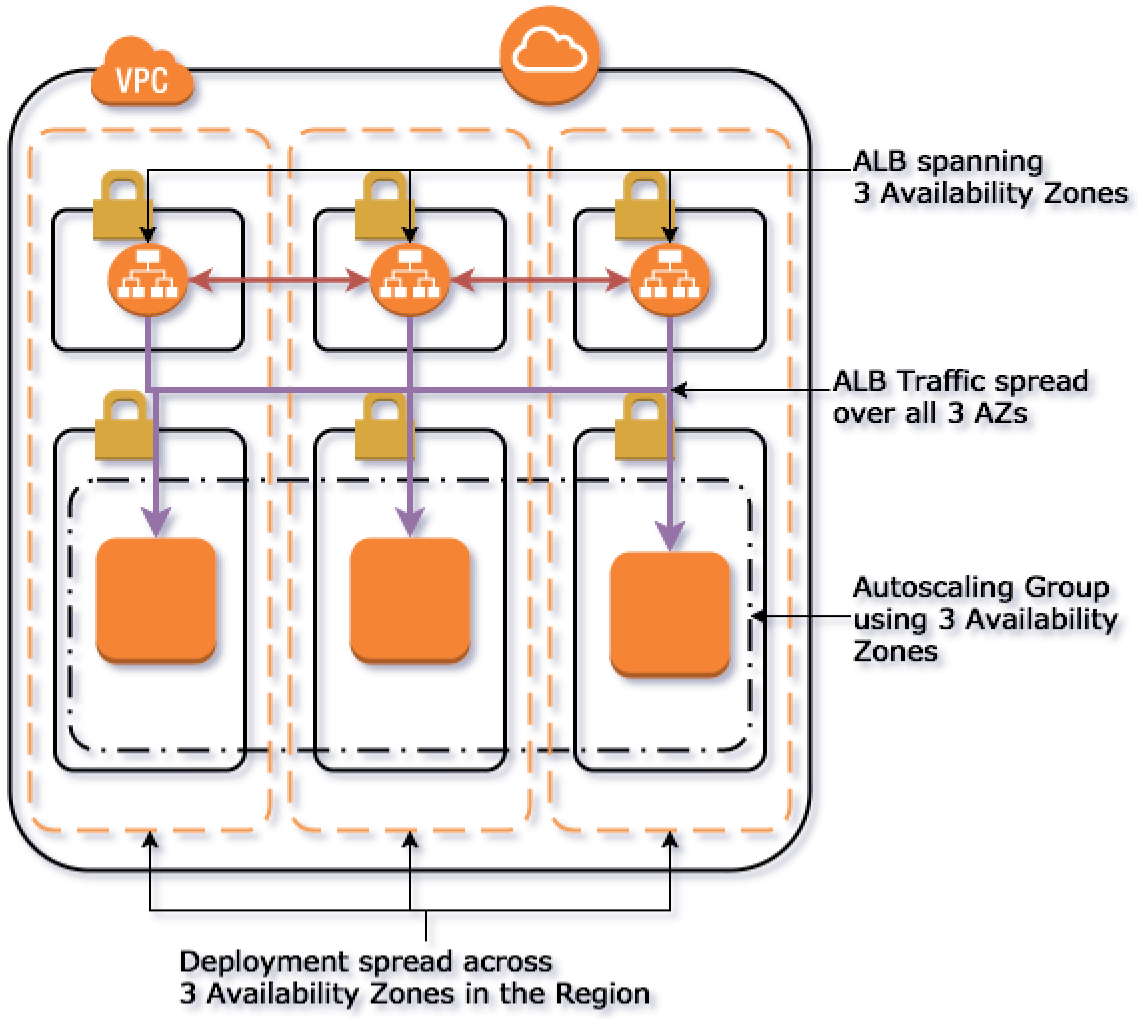

VPC
ALB spanning 3 Availability Zones
ALB Traffic spread over all 3 AZs
Autoscaling Group using 3 Availability Zones
Deployment spread across 3 Availability Zones in the Region

FIGURE 11 - Leverage AWS services like ALB and Autoscaling Groups to further add resiliency to your service.

First, an ALB is setup to span the three availability zones in the deployment. This ensures that if there is a localized problem with one of the availability zones, the impact to overall service is minimized. Second, as indicated by the purples arrows in Figure 11, traffic from the ALB is balanced across all available instances. These are examples of the redundancy (as opposed to single instance) principle already discussed. And, as your network becomes more complex, you'd want to manage all of this using CloudFormation, which really embodies the automation and repeatability also discussed. (We'll talk more about CloudFormation in the next section.)

Add Health Checks to Your Instances

There's another key ingredient that's not as obvious from Figure 11, but remains essential. That's a health check that each instance (or in some cases, each service or application running on each instance) provides. A *health check* is usually an API endpoint or other easily-checked status code that says, "Yes, this instance is good to accept requests" or, when non-responsive, indicates, "Sorry, something has gone wrong with this instance."

Health checks typically are scripts that have certain steps that take to verify that an instance is working. They're robust and don't give a lot of false negatives. When a health check fails, your system can shut down the failing instance and replace it with a fresh one. (Here's where repeatability and automation are crucial!)

Load is Another Form of a Health Check

In the same vein as the health check, the load on an instance is an indicator as to whether traffic should keep flowing to that instance. While an instance with a particularly high load doesn't need to be recycled like an instance with a failing health check does, it still is a sign that traffic should (at least temporarily) be halted to that instance. CloudWatch can give you an up-to-the-minute check on CPU and network utilization, and trigger alarms when thresholds are reached that indicate high load.

What's even better than a simple stop of traffic on high load is converting that to an indicator for scaling up. When load reaches a certain point, new instances should be added to the system. When load lowers enough, instances can be turned off and shut down. And because you've got everything automated, this all happens invisibly–without human intervention.

Create and Maintain Your AWS Fingerprint Using CloudFormation

With bastion sub-accounts, multiple private VPCs within accounts, segmentation for billing, and IAM policies, there is a lot to keep up with to manage a functional and secure AWS network. In the same way that a cloud platform can reduce costs associated with maintenance of physical servers and networks, the platform can also reduce the deployment and creation costs. AWS provides several tools for automating these processes, most notably AWS CloudFormation (https://aws.amazon.com/cloudformation/). CloudFormation allows you to manage your infrastructure as code.

CloudFormation is also a key ingredient in the scalability just discussed. If every time an instance failed a health check or reached load capacity, an engineer had to manually create and deploy a new instance, you'd not only need to increase your staff, but keep them working 24/7. That's hardly the promise of the cloud. CloudFormation is a repeatable, runnable "snapshot" of a set of tasks: to start a new instance, to deploy an entire network complete with VPCs, to configure a database instance and connect it to a cluster.

The goal of a good AWS environment should be to have every single resource managed by CloudFormation. In addition to enabling elasticity and scalability, this provides traceability and accountability. That's a lot of "ilities," but CloudFormation is a version-controlled resource that tells you what is deployed, who last updated and made changes, and (ideally) explains why those changes were made. Add to this the ability to peer review CloudFormation and you have a sizeable advantage over systems with collections of system-level shell scripts with minimal change history or version control.

While CloudFormation is really an entire white paper unto itself, you can get a taste of CloudFormation by visiting AWS's Getting Started tutorial online (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/GettingStarted.Walkthrough.html). Once you have the basics down, take a look at the CloudFormation template included with the lambda function that deregisters hosts from the FreeIPA directory server earlier in this paper (see the section Leaving a Directory Server). That will give you a good idea of what a CloudFormation template looks like in our own infrastructure.

Conclusion

Cloud computing platforms–and AWS in particular–are changing the landscape and profile of development and operations within organizations large and small. IT is becoming less of an organizational division and more of a service broker. With this change comes a growing expectation for reliability, scalability, and an accessible platform for innovation.

There are currently nearly 120 discrete AWS services, and more are added each month. While traditional IT suggests that the best policy is restrict as much as possible, that is not the approach of this paper or effective organizations using the cloud. Instead, a balance of compliance, responsibility, automation, and security provide the flexibility that development teams crave while still giving organizations a secure production environment. There will always be regulatory or policy issues that cause particular services to be restricted, but that should not be the default position for a growing and innovative organization.

The challenge, then, is to take the lessons from this paper and add to them your own organization's learning. Make informed decisions that serve your organization's needs rather than a particular IT or departmental tradition. And, of course, Element 84 is always interested in talking to you and assisting with your cloud needs. Reach out to us at info@element84.com and we are happy to help you with your infrastructure and cloud requirements.

Appendix 1: Further Reading

The following URLs provide relevant and helpful information on managing AWS environments.

- Best practices for Security in AWS - Amazon White Paper: https://d1.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf
- Managing the Transition to IT as a Service Broker - Rackspace White Paper: https://api.built.io/v1/classes/Listing/objects/blt49844e776bfb6610/uploads/57ffcc62f475e56808bdd4c0/application_api_key/bltfb154434f96eaff4
- Listing of all current AWS services: https://aws.amazon.com/products/
- Report on current cloud market share of major vendors: https://www.cnbc.com/2018/04/27/microsoft-gains-cloud-market-share-in-q1-but-aws-still-dominates.html
- Top 7 Reasons to Migrate Your Data Center to the Cloud - Apps Associates: http://www.appsassociates.com/downloads/Top-Seven-Reasons-For-Migrating-Your-Data-Center-to-the-Cloud-E-Book.pdf
- Setting up Federated SAML login to AWS from Google Apps: https://aws.amazon.com/blogs/security/how-to-set-up-federated-single-sign-on-to-aws-using-google-apps/
- Reasons Why Shadow IT is starting to emerge from the Shadows: https://www.zdnet.com/article/6-reasons-why-shadow-it-is-emerging-from-the-shadows/
- The Hidden Truth Behind Shadow IT: http://digitaltransformation.frost.com/files/4313/9300/1515/rp-six-trends-security.pdf
- An explanation of DMZ networks and their role in security: https://fedtechmagazine.com/article/2017/07/what-dmz-network-and-how-can-it-improve-your-security
- FreeIPA Open Source Directory Server: https://www.freeipa.org/page/Main_Page

Contributing Authors

The following folks contributed to the creation of this white paper, whether through research, writing of content, editing, or graphic design.

- **Kevin Mentzer**
  Sr. DevOps Engineer, Element 84
- **Dan Stark**
  Sr. DevOps Engineer, Element 84
- **Brett McLaughlin**
  Principal Solutions Engineer, Element 84