DATE: [redacted]
TO: [redacted]
FROM: Omar Brown
SUBJECT: Deploying Doors on AWS - CodePipeline Development

Introduction

Over the last week or so, I had been tasked with researching and testing a deployment of the [redacted] Application to the AWS platform. I have successfully deployed our java application, report server, and database onto the amazon web services platform, as well as created a plan for implementing our code pipeline. This move will allow us as developers to test and deploy changes to the product quickly, efficiently and automatically. This report details the needs of the project, the architecture of the application, setting up a CI/CD pipeline, and an analysis of the costs.

Needs of the Project

When planning a deployment architecture, many different needs must be taken into consideration. Architecture design choices can be very different depending on the needs of the development team, the application, and the client. In the case of [redacted] Project we must consider the following:

A. The company is mid-sized and end-user will be connecting primarily from one or two locations
B. The application is inward facing (for company use only)
C. The bulk of the application runs on tomcat, with a database and a business intelligence server supporting it
D. The development team is flexible and small
E. Most database queries are OLTP and not OLAP

These needs allow us to focus on a design that is smaller, but able to automatically scale outward during times of peak usage. We can implement a CI/CD pipeline for an agile development process, and automate Talixa's deployment process. The database initially does not need to implement read-replicas or caching solutions. We may implement those solutions in the future after we see how our application acts out in the wild. For the beginning, we can configure our database system using the Multi-AZ feature to ensure that redundancy is built in (see *Elements of the AWS Architecture - RDS.)*

The Architecture

Design Overview

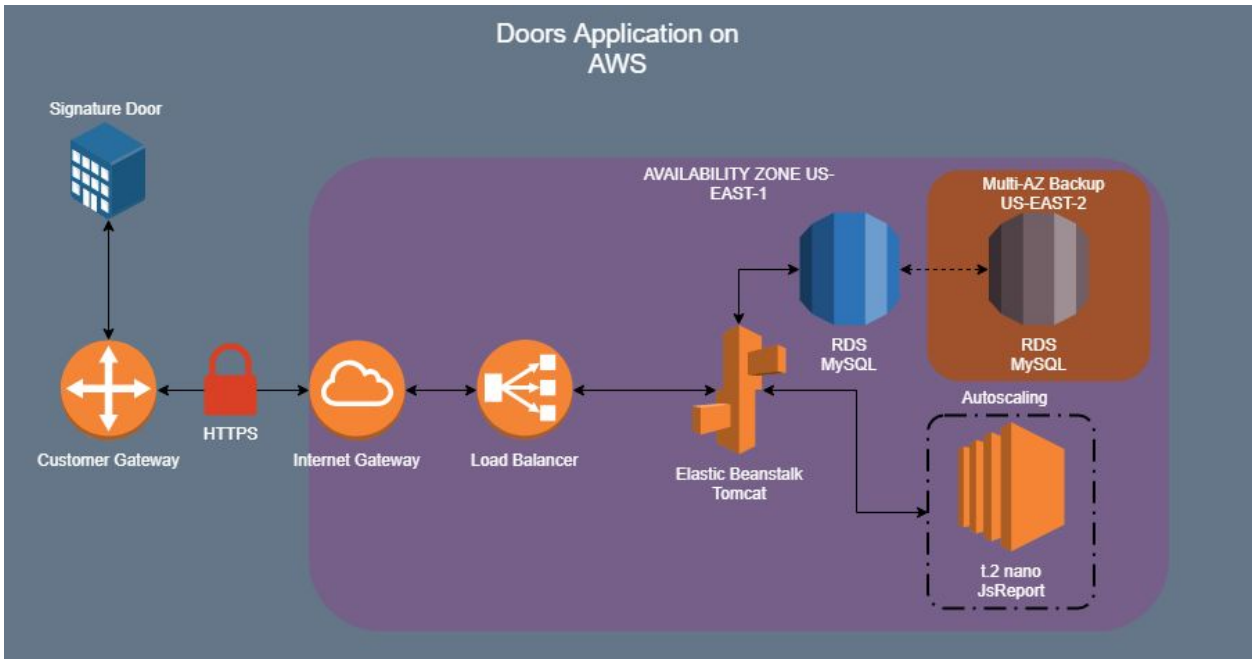

Figure 1: Doors Architecture on AWS

The diagram above details the design and architecture of the production application running on the AWS platform. The client will be able to connect to the application server via HTTPS. Our application server will communicate with both our reporting software as well as the database inside the same VPC (see below).

Elements of the AWS Architecture

[redacted]

This part of the application is the most important, the end user. The customer will be connecting to their gateway via their local IP address.

Customer Gateway

This is the router of [redacted], and the device that gets a public IP address. This public IP address will be whitelisted on the AWS VPC gateway for security. Only public IP addresses that are whitelisted will be allowed to connect to the application.

Availability Zone US-EAST-1

This box is meant to show the VPC living in the availability zone US-EAST-1 (W. Virginia). A VPC is a separated network within AWS for our environment.

Internet Gateway

This device allows traffic to and from the outside world to communicate with devices inside the Virtual Private Cloud (VPC). The Route Table settings dictate what traffic can leave, and traffic is controlled by Network Access Control Lists, as well as the security group each component has attached to it. To limit traffic for all connections in, place a rule in the Network ACL.

Load Balancer

This network device is set up as a network load balancer. It allows for autoscaling at the elastic beanstalk component. When latency gets high, elastic beanstalk will scale up and the load balancer will route traffic between instances.

Elastic Beanstalk

This service allows us to deploy and manage our web application. It will automatically scale and has configuration files for each instance that is sprung up. The config files currently fetch the needed tomcat libraries from an s3 bucket and place them in the needed folders.

RDS

This is where the database for our web application lives. It allows us to avoid setting up/installing/maintaining database software and can be interacted with via the mysql shell or aws console. It is highly scalable with multiple features being available for data caching, replication, and backup management. In our case, we can use the Multi-AZ feature to create a read-replica standby in case of instance failure.

Multi-AZ RDS

This feature is for redundancy, not availability. AWS manages a read-replica of our database in another availability zone. If our RDS instance in US-EAST-1 fails, AWS automatically routes the original endpoint to our backup. This provides assurance that if one fails, our application remains available.

EC2 t.2 nano

This is our jsreport server running on EC2. To save costs, we are running it on a nano sized instance which will autoscale just in case it is needed. It can be interacted with (as all resources can via ssh or aws console)

Conclusion

This architecture allows us to have a small application set up in AWS. If any component of the application needs to scale out, it will do so automatically without intervention by the developers. The database is set up with redundancy in mind, and other features can easily be implemented later on. The client connection will be secure over HTTPS, and the different components can be managed by our on-staff AWS Developer.

DevOps and CI/CD Pipeline

Introduction

Modern DevOps is a very powerful tool and by hosting our application on AWS we have the ability to implement a CI/CD pipeline for this application. This will simplify adding features and future application management both during testing and production. It will improve code quality, improve developer productivity, and automate our delivery process. According to the AWS Whitepaper titled *Practicing CI/CD on AWS*, it is recommended to transition in small steps to a fully automated delivery process. For the first step, we will implement a Continuous Integration pipeline. As we progress, we can work towards a fully matured Continuous Deployment process.

DevOps Best Practices

Before defining the different components of our pipeline, I think it is highly important to note that this is a change, not only in our build process, but also in our approach to commiting changes and developing. I highly recommend reading the whitepaper from AWS, which you can find in the references section. Some important best practices from that paper are outlined below, as well as how they relate to us:

1. Treat your infrastructure as code
    a. We will need to move on to a cloudformation template for this entire pipeline so that we have our infrastructure outlined as code. I would like to request a new ticket to be opened that I may work on to completely draw up our infrastructure for Stage 2 of our automated delivery process.

2. Keep teams small
    a. As Talixa continues to expand, it is recommended to keep developer teams small

3. Make frequent and small commits
    a. Avoid large changes of the codebase being committed all at once
    b. No long-running feature branches

4. Make unit testing 70% of the overall testing

5. Keep metrics
    a. How long does each part of the pipeline take?

6. Use separate pipelines for each team

In conclusion, we can use these best practices to make our team more efficient, more organized, and automate our entire deployment process from commit to end-user. Detailed below is the first stage of our transition to a pipeline.

Stage 1: Continuous Integration Pipeline

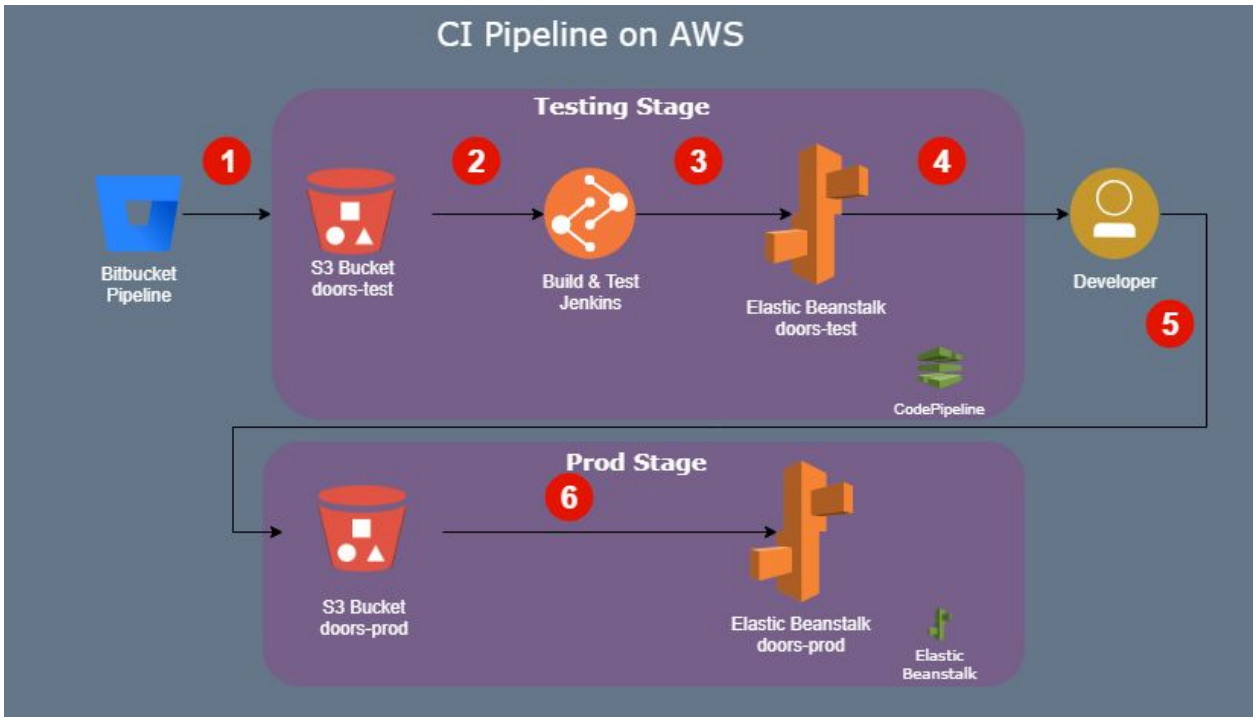

Figure 2: CI Pipeline for Doors Project

The diagram above is a visual representation of our pipeline setup for the first stage of moving to CI/CD. There is only one process that is done by a developer (5) and everything else is automated and tracked within AWS. After this stage is implemented, it can be defined as code using CloudFormation.

Testing Stage

1. After a commit is made to bitbucket, the basic build pipeline starts. Once the changes pass in the Bitbucket Pipeline, a new version of the package is created and moved to an S3 bucket (this is implemented in the bitbucket console, and natively supported). The S3 bucket holds a .war package of our codebase, and has versioning enabled. This update of the codebase triggers an event that then initiates CodePipeline to begin.
2. CodePipeline moves this package into our unit testing stage. This is managed by Jenkins in AWS (or our current jenkins server).
3. Once everything passes our unit tests, CodePipeline moves our codebase into a test environment powered by Elastic Beanstalk.

Developer Interaction

4. A developer reviews the test application in the test environment

Production Stage

5. Once approved moves the .war file from the S3 test bucket to the S3 prod bucket. The command to do this is:
   `aws s3 cp s3://doors-test/doors.war s3://doors-prod/doors.war`
6. The S3 prod bucket also has versioning enabled. This update of the package triggers our production environment to update automatically and deploy the changes.

Conclusion

In summary, we can move the project into AWS to implement DevOps practices. This will allow us to produce better quality code, make changes to production quickly, and become more productive. Our development process can implement the first stage, a CI pipeline, before moving onto stage two. The second stage would be to move our infrastructure over to code, and automate the entire delivery process eliminating step four of our CI pipeline.

Cost Analysis

Introduction

Hosting our application in the AWS environment has many benefits for us and the customer, but we must ensure we find the most cost-effective solution. AWS has various pricing models, each with distinct advantages and disadvantages. I have quoted us using different pricing models and I think we should start with on-demand pricing for the first three months of moving to AWS so that we can accurately assess the needs of the different servers. I would like to set up the environment and then realistically load test the different components to ensure a smooth production environment.

After moving over to AWS and thoroughly assessing our company's needs, as well as the needs of the client, we can look at paying for reserved instances. Here are the key differences between on-demand and reserved instance pricing models:

On-demand:

- Pay-As-You-Go model
- Highly flexible
- More expensive

Reserved:

- 1yr or 3yr contract
- Much cheaper
- Can be made more flexible with Reserved Convertible, but not like On-Demand
    - Reserved Convertible lets us change the instance type (for example a t2.small general purpose to an h2.small high throughput instance)
- Partial allows us to make a small partial upfront payment for more savings, and All-Upfront gives us even more savings by paying the entire cost in one upfront payment

Spot:

- This option is for bidding on server time
- Great for certain use-cases like mapping the human genome or running weather simulations
- Is extremely cheap, but has to run for a predetermined amount of time

Quotes

Based on the architecture design, I have generated several quotes for the cost of running the Doors Application and CI/CD pipeline in AWS. Below are three different pricing models, and the first-year/annual costs.

On-Demand

Role	Instance	Hourly Cost	Up-Front Cost	Yearly Cost (24/7)
JsReport Server	Ec2 Linux Nano	$0.006	-	$52.56
Test Server	Ec2 Linux Micro	$0.012	-	$105.12
Prod Server A	Ec2 Linux Small	$0.0230	-	$201.48
Prod Server B	Ec2 Linux Small	$0.0230	-	$201.48
Load Balancer	Network	$0.025	-	$219.00
RDS	Db t2.medium Multi-AZ Durable	$0.068	-	$595.68
Annual Cost				**$1375.32**

Reserved Pricing 1 Year (Partial Upfront)

Role	Instance	Hourly Cost	Up-Front Cost	Yearly Cost (24/7)
JsReport Server	Ec2 Linux Nano	$0.003	$15.00	$26.28
Test Server	Ec2 Linux Micro	$0.007	$30.00	$61.32
Prod Server A	Ec2 Linux Small	$0.014	$60.00	$122.65
Prod Server B	Ec2 Linux Small	$0.014	$60.00	$122.65
Load Balancer	Network	$0.025	-	$219.00
RDS	Db t2.small Multi-AZ Durable	$0.022	$204.00	$192.72
First-Year Cost			**$369.00**	**$744.62**
Annual Cost				**$1113.62**

Reserved Pricing 3 Year (Partial Upfront)

Role	Instance	Hourly Cost	Up-Front Cost	Yearly Cost (24/7)
JsReport Server	Ec2 Linux Nano	$0.002	$30.00	$17.52
Test Server	Ec2 Linux Micro	$0.005	$66.00	$43.80
Prod Server A	Ec2 Linux Small	$0.009	$122.00	$78.84
Prod Server B	Ec2 Linux Small	$0.009	$122.00	$78.84
Load Balancer	Network	$0.025	-	$219.00
RDS	Db t2.small Multi-AZ Durable	$0.0162	$436.00	$142.11
First-Year Cost			**$776.00**	**$580.11**
Annual Cost				**$580.11**

Conclusion

Based on my current understanding of the clients needs, as well as an understanding of what we need as a company, I recommend the following:

- We use on-demand pricing for the first 4 months of the switch to AWS to accurately assess the needs of our testing environment and see how it behaves “out in the wild”
- We implement a CI/CD pipeline in AWS using the resources we would need for the testing stage of the application.
- Once we have this testing environment polished to perfection, I then recommend purchasing the 3-yr reserved since we will need this environment for the lifetime of the application.
- Once a minimum viable product is finalized, we repeat the same steps as above for the production environment (test our architecture on-demand, then reserve resources)

This approach allows us to validate the needs of our application, our client, and our team. Once the environment is polished and tested, we then reserve for three years saving us and the client money.

Summary

In this paper we discussed the needs of our application, the architecture on AWS, setting up our pipeline, and the costs associated with making this change. I think that this move will bring great value to our ability to maintain and scale this enterprise application. With DevOps best practices and an automated pipeline, we can make small feature changes regularly with ease. This move will also increase developer productivity, and improve application redundancy and scalability.

AWS has a lot to offer us as a team, and I look forward to implementing the power of the cloud to our project.

Resources

AWS CI/CD Whitepaper
https://d1.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

AWS Pricing
https://aws.amazon.com/pricing/?nc2=h_ql_pr