Continuous Integration in the Cloud Proposal

Vision and Goals of the Project:

Continuous Integration in the Cloud will be providing users (programmers) with Continuous Integration services using pre-configured environments and platforms build in the cloud, pulling down an image from the cloud to users' laptops. High-level goals of Continuous Integration in the cloud include:

- Providing a simple, straightforward user experience for programmers with little to no set-up required
- Providing programmers a way to test their code in the environment that the code will be put into production on
- Enabling a programmer to automatically test, push, and deploy his code into production as well as see these changes in real-time.

Users/Personas of the Project

Continuous Integration in the Cloud will be used by program developers, companies, and server administrators.

It targets only programmers and server administrators who will be providing the image to programmers.

It does not target:

- New programmers with little experience in continuous integration.
- Companies who do not use continuous integration technologies like Jenkins and Docker.
- Anyone without knowledge of continuous integration techniques.

1. Scope and Features of the Project:

Continuous Integration Image

- Presents a service that could integrate the whole process of DevOps from software development to production and then push it into the cloud.
    - Multiple OS and programming language support.
    - Load testing in the cloud.
        - Dockers and Jenkins techniques included
        - Vagrant development environment required.
        - Real-time updating on the target website.
        - Multiple-access support

2. Solution Concept

Global Architectural Structure of the Project and a Walkthrough:

Below is a description of the system components that are building blocks of the architectural design:

- Jenkins: Continuous Integration tool that keeps track of a Git repository and triggers by a build to a project with code changes.
- Docker: Portable container technology that creates an individual "container" for specifics parts of a project. These containers are configured to run for example, front end code, back end code, Jenkins jobs, and test suites. Docker also enables us to push these containers from environment to environment without having to recreate them, so for the scope of this project, from local machines to our QA Environment in the cloud.
- Vagrant: An application that sits on top of a VM that creates a development environment that can be consistent from system to system. For the scope of this project, we will use Vagrant on our local machines and in the QA Environment to set up a consistent product environment so as to guarantee that code is built and run in a consistent environment.
- VirtualBox: A free tool that allows users to run VMs on their machines. We will run vagrant on top of this to maintain continuity from environment to environment.
- Linux based OS: We have not decided exactly on which type of Linux we will use. It will most likely be either Project Atomic (a Linux build that was made for Containerized DevOps), Ubuntu (most widely used free Linux distribution), or CentOS (widely used in industry for DevOps).
- Apache Maven: A tool that can manage the build process of a project based on its POM file. Once a Jenkins job is executed after a code check in, a Maven instance will be called to automatically build the Web App.
- Selenium Webdriver: An automated testing API that imitates a user interacting with a web browser. Selenium can click through pages, verify text and images, form filling and submissions, etc. Has support for Java, Python, C#, and Ruby. Will be used for functional testing.
- Java or Python: High level programming languages we are most familiar with that we will use to write scripts for functional and load testing.
- Amazon EC2: While this could be changed, we will need some type of web server for our Cloud Environment and as EC2 is the most widely used, this will most likely be the option we choose. Truth be told, we haven't put much thought into this aspect and need to research it more.

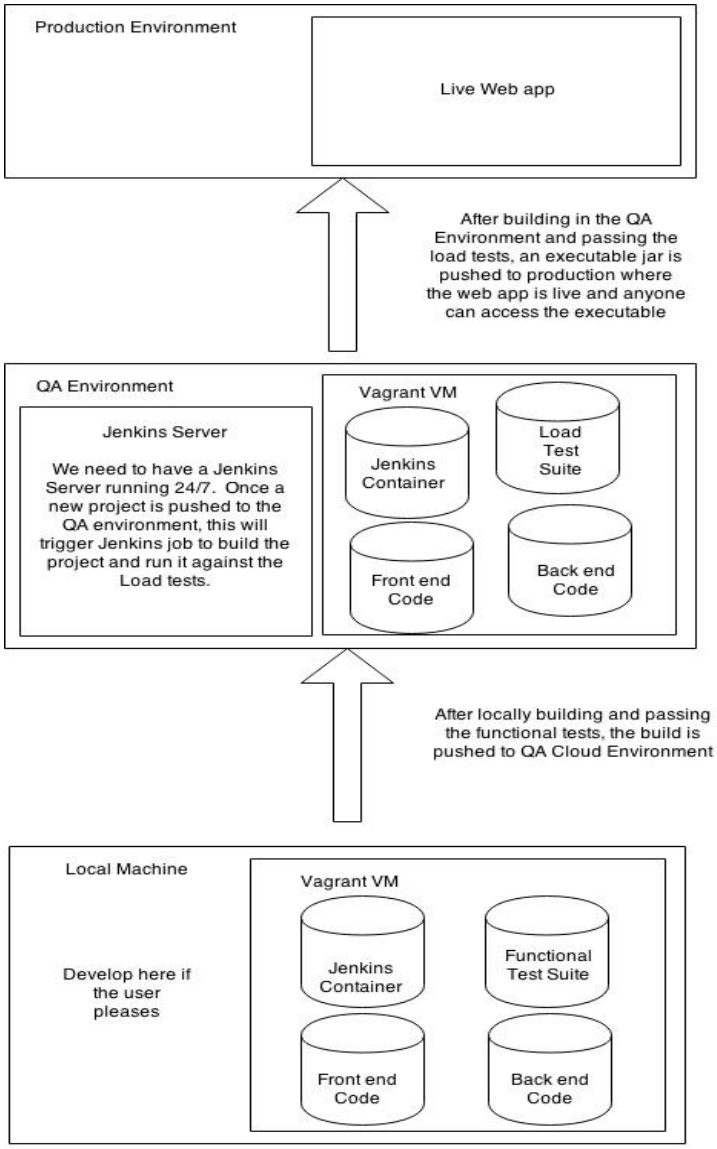

Figure 1: Continuous Integration Image architecture.

Figure 1 presents our global architectural design for the Continuous Integration in the Cloud Project. The bottom level is the developer's local machine. At this level, the developer can develop in any base OS in the environment they're most comfortable in. After they have finished code changes, they can open up their VirtualBox Linux VM with Vagrant running on top of it. Within this Vagrant VM, there will be individual Docker containers, one each for the Front end code, Back end Code, Functional Test Suite, and Jenkins Job. After they are in the Vagrant environment, they can push the code changes to their local repository where a Jenkins job will triggered where it will call Maven and build the project. If the project builds successfully, it will then be run against the Functional test suite. Once it passes the functional test, the code changes will be pushed from the developer's local repository to the Git repo in the cloud.

The middle level is the QA Environment in the Cloud. Once the code has been pushed to the Git repo, the Vagrant VM in the cloud will hold four containers like on the local machine. Three will be the same: The Back end, Front end, and Jenkins Containers, however, there will be a Load Testing Container instead of functional testing because we determined the Web App already works, at this point it needs to be tested to ensure it can handle the appropriate amount of users. The process of getting it to the point of testing and deploying to the Production Environment is the same as it is on the local machine to deploy to QA. This level also includes a dedicated Jenkins server. Since this project will be used on a wide scale, it is necessary that there be a Jenkins server live 24/7 so users can create and execute Jenkins jobs within their Vagrant environments for their particular project.

The top level is the Production Environment. Once the project is built and load tested in the QA environment, the live version of the Web App will be pushed into production and live for anyone with permission to access.

Design Implications and Discussion

Key design decisions and motivation behind them.

- Jenkins: While there are many CI tools, Jenkins is the most widely used in industry. For the scope of this project, Jenkins is the best option because it has plug-ins for Docker, Maven, and Selenium, allowing us to implement these technologies with Jenkins with little to no issues.
- Vagrant: A huge issue in industry today is that developers write code and test on their local machines and it works fine. Yet, when they push it to their company's QA Environment, it breaks due to different libraries and dependencies for programming languages in different programming environment. Vagrant fixes this problem because it creates an environment configuration that can be migrated from machine to machine or environment to environment, allowing continuity from stage to stage in the development, testing, and deploy process.
- Docker: We are choosing to use Docker to create containers for applications, test suites, and Jenkins jobs because it allows developers to manage and track dependencies for the particular project within each container. Since all dependencies and artifacts are stored in this container, it can be moved from environment to environment without any issues and can therefore work on any platform.
- Apache Maven: Maven will be used to automatically build projects once pushing code changes to a repository triggers a Jenkins job. Maven is being used because the build process is configured in the projects POM file which is created in almost every

application written using a high level programming language for large scale distribution or use.

- Linux OS: A Linux distribution will be used as it is industry standard for DevOps environments and SysAdmins. The exact distribution is still being decided as we are weighing the pros and cons of Project Atomic (mainly used for Containerized DevOps), Ubuntu (Most used distribution), or CentOS (widely used in industry for DevOps).
- Amazon EC2: As stated earlier, this is the most widely used Web Server in the industry. Because of this, there is the most documentation and support online as opposed to other options. We haven't put enough time into the decision process to decide upon this definitively.

Acceptance Criteria

Minimum acceptance criteria is a being able to pull code directly from a user's computer, test the code within the production environment with Docker, push the code directly to the cloud once all the Jenkins tests are successful, run the same code again inside the cloud with more intensive Jenkins tests, and finally push the code into production and see changes in real time. Stretch goals are:

- Do the above process with multiple languages and operating systems
- Use some codes and sample to show the working process
- Construct a single universal image that can test most code on most operating systems
- Develop an easy to use UI for users to specify the code being tested, which operating system the code should be tested on, and the tests that should be run on the code within the containerized version as well as the cloud

Release Planning

Detailed user stories and plans are on the Trello board:
https://trello.com/b/TNhinPsC/continuous-integration-in-the-cloud

Release #1 (due by Week 5):
Local Machine Environment:

Have the local machine setup with VirtualBox running a Linux VM with Vagrant on top of it. Within here, we will have the 4 Docker containers, Front end Code, Back end Code, Jenkins Container and Functional Test Container. Technically, the Jenkins Container and the Test Container will be empty at this juncture, but it will exist so the Test Suite can be developed for the next demo and a Jenkins job will created for its container as well.

Release #2 (due by Week 7):

Jenkins Test Suite/Environment:

On the local environment, we will have created a Jenkins job that will check when code changes have been pushed to the developer's local repository. When the push occurs, the Jenkins job will be triggered which will call Maven to build the project and upon successful build, will run the test scripts to test functionality of the web app.

Release #3 (due by Week 9):

QA Environment/Jenkins Server:

We want to have the QA Environment functional to the point where we have a VitrualBox Linux VM with Vagrant running on top of it. We will once again have four docker containers containing Front end code, Back end code, a Jenkins Container, and a Load Test Suite. Once again, the Jenkins Container and Test Container will be empty, but will be created so we can write the load tests and create the Jenkins Job.

We will also have the dedicated Jenkins server up and running. This server will be live 24/7 so anyone using this Open Source CI tool can create and execute Jenkins jobs at their discretion.

Release #4 (due by Week 11):

QA Jenkins Test Suite/Environment:

At this point, code should be able to be pushed from the local repository to Git repository in the cloud. This push will trigger a Jenkins job in the QA environment that will call Maven to build the project and upon a successful build, will test it against the Load Test Suite to make sure the Web App can handle a substantial user base. Upon successful completion of the load tests, the executable jar will be pushed to production.

We will also have a fully function Production Environment which will be an EC2 Web Instance that will host the live version of the Web App.

Release #5 (due by Week 13):

Complete Release:

By the end of week 13, our project should be complete. The image will be able to be pulled from the cloud, have a fully functional and easy to use UI to prompt for the code being pulled directly from the computer its being programmed, run simple tests within the proper OS, push the code to the cloud server if there are no problems or prompt the user if there are, continue to test the code more rigorously in the cloud as well as notify other members of the

code changes requiring their commit, and finally pushing the code into immediate production if all tests are successful as well as all members committing on the code for a real time update.