Hand-in assignment 2 – Docker in the Cloud

Maciej Owczarek 579390, Marcin Grawiński 579370

# Meeting the Learning Objectives

### **Dockerize an existing service or application**

The CloudShirt .NET solution has been containerized. Its Dockerfile (or equivalent containerization configuration in the repository) ensures that the application runs within Docker containers.

### **Deploy a Docker application on a server in the cloud**

The CloudFormation templates provision AWS EC2 instances (both the build/master server and the worker nodes) that are preconfigured to install Docker, download the CloudShirt code, and run docker-compose. This demonstrates how a Docker application is deployed on servers in the cloud.

### **Deploy a Docker cluster over multiple servers**

In our solution we use docker swam. The build instance is configured as the Docker Master. The master instance initiates the swarm and generates the join token, which is stored and shared (via the EFS mount) with the worker nodes. AutoScalingGroup instances join the swarm, forming a multi-node cluster.

### **Provision a private Docker registry and push a Docker image to this registry**

A private Docker registry is provided by AWS ECR. The build (master) instance logs into the registry, builds the Docker image of the CloudShirt application, tags it using ECR URI, and pushes it to the repository.

### **Deploy a dockerized application on a Docker cluster from a private registry**

The worker instances in the Auto Scaling Group pull the Docker image from the private ECR repository during their startup. They then run docker-compose to launch the containerized CloudShirt application, ensuring that a dockerized application is deployed on the Docker cluster using images stored in the private registry.

# Meeting the Requirements

### **REQ-08 "The CloudShirt .NET solution should be dockerized"**

The solution repository (CloudShirt) is dockerized, as evidenced by its container build and docker-compose configurations used in the templates.

### **REQ-09 "The CloudShirt .NET solution should be built from a separate AWS EC2 instance in the private subnet, called Buildserver"**

Buildmaster.yml file includes a BuildInstance resource deployed in a private subnet. This instance is responsible for running nightly builds, performing git pulls, compiling the application, building the Docker image, and pushing it to the private registry (ECR).

### **REQ-10 – "Docker-compose is used to define and run the services to be deployed to the instances in the ASG"**

Both the build (master) instance and the worker nodes use docker-compose configuration files that define all required services and their dependencies, ensuring consistency and ease of orchestration.

### **REQ-11 – "The CloudShirt .NET Docker images should be built on the Buildserver during nightly builds"**

A scheduled task (implemented using systemd timers) on the build instance triggers nightly builds. This procedure rebuilds the Docker images and pushes updated versions to the private repository.

### **REQ-12 – "The CloudShirt .NET Docker images are pushed to an AWS ECR or Docker.io private repository during the nightly builds."**

The build process includes commands for logging into AWS ECR, tagging the built Docker image with the repository URI, and pushing the image.

### **REQ-13 – "The Buildserver is configured as master in a Docker cluster."**

Upon startup, the BuildInstance initializes Docker Swarm (using docker swarm init), generates a worker join-token, and writes necessary data to the shared EFS mount. This configures the buildserver as the master in a Docker cluster.

### **REQ-14 – "The instances in the ASG are configured as workers in a Docker cluster."**

The Auto Scaling Group is associated with worker nodes that, during their startup user data script, are configured to join the Docker cluster (using the join token and master IP provided via EFS).

## Most Important Choices

In this solution we wanted to setup a pipeline. Where software engineers would develop application and everyday at 3:00 am new image would be build. There were two ideas –  
First to make the buildMaster instance the only source of production version of the app.

This would simplify every next build as the github repository is not developed towards cloud application.  
Second to make CloudShirt github repo the only source of production version of the app. This makes working on the code more straightforward for developers. It also provides better version control.   
We chose the second option as it better aligns with operational excellence of WAF. To anticipate failure and treat resources as disposable means that the buildMaster instance will terminate sooner or later and then nothing of importance should be lost.

To access the dashboard from our private subnet, we evaluated several options including a bastion host and VPN. At the end, we decided to choose AWS Session Manager port forwarding because it provides secure access to the Kibana dashboard via a web browser on localmachine without the limitations of a bastion hosts and it is also a convenient and simple solution.

We also needed a way of transfer the swarm token from buildMaster instance to worker instances. We chose EFS as it was already used in our solution.

# How to Roll Out the Solution

To rollout the solution first download all files or clone GitHub repository (<https://github.com/Grawikos/CloudAutomation.git> no-terraform branch). The lab environment is expected, otherwise, a LabRole with LabInstanceProfile must be created. AWS CLI must be installed and set up with default region and credentials. In the folder with all files execute

|  |  |
| --- | --- |
| Windows | MacOS/Linux |
| *create.ps1 [-userid “<account user id>”, -bucketname “<name>”]* | *deploy.sh [-userid “<account user id>”, -bucketname “<name>”]* |

*<account user id>* - optional, default: found by command  
*<name>* – optional, name used to create a bucket. It will be concatenated with userid to decrease chance of name collision, default: “athena-data-bucket”

After all stacks are created, the link to the website will be outputted to the terminal. It may take a minute to initiate servers.

To deploy static website on the S3 Bucket go to folder with bucketscript.ps1 and website.json and execute.

|  |  |
| --- | --- |
| Windows | MacOS/Linux |
| *bucketscript.ps1 [-userid “<account user id>”, -bucketname “<name>”, region “<region>”]* | *bucketscript.sh [-userid “<account user id>”, -bucketname “<name>” , region “<region>”]* |

*<region>* - region of the user, default: us-east-1

The link to the website will be outputted to the terminal.

To connect to the ElasticStack and see logs of the application as admin, execute.

|  |  |
| --- | --- |
| Windows | MacOS/Linux |
| *connect\_admins.ps1 [-port “<local Port Number>”]* | *connect\_admins.sh [-port “<local Port Number>”]* |

*<local Port Number>* - any open port on the host, default: 8080.

The link to the website will be outputted to the terminal.

On Kibana website (default - localhost:8080), a new dashboard needs to be created in the Discover tab with logs tagged as filebeat (e.g., filebeat-\*).

## Recommendations

### **Better CI/CD pipeline**

Most changes to the code should be done in github repository, but at the moment application on github is made for local usage only where the buildMaster instance deals with translating it to Cloud environment. This is error prone and it would be better to update the github repo or to create a new one with Cloud integration applied and simplify the buildMaster docker build script.

### **Avoiding Single Points of Failure**

The RDS is crucial to the application, so having only one instance without any redundance is a bad practice and should be fixed by making copies of it.

### **Move application session storage to a database**

Right now the application stores a session in memory and a load balancer keeps track of which client should go to which instance. This is bad as if an instance is deleted, the session will be terminated.

### **Secure Access & Data Protection**

Implementing user authentication for accessing the S3 bucket and encrypting data in both the S3 bucket and EFS would improve security measures.

### **Optimizing Performance**

Employing a caching solution, like AWS CloudFront, to provide static content would reduce the load on backend systems and improve user experience.

### **Implementation of Policies & Roles**

Defining reliable IAM policies and role-based access controls is crutial. Although the current lab environment limits us from creating these policies, but their implementation would significantly improve security and operational management.