AWS deployment of a virtual laboratories system

Kraus, Arkadiusz
akraus@student.agh.edu.pl

Naróg, Mateusz
narog@student.agh.edu.pl

Wójcik, Anna
annawojcik@student.agh.edu.pl

Zoń, Sylwia
zon@student.agh.edu.pl

March-June 2021

1 Introduction

As a Bachelor Thesis [1] there was created a system that supports automated creation and management of virtual laboratories, which is a provision virtual machine or container with a software that is needed for a specific task. System initially was supposed to prepare a virtual environment for programmable networks. However, during the work it was generalised to support lab prepared as a container or virtual machine.

The natural continuation of this work is a deployment of this system. Also a lab on programmable networks was needed for a different subject to conduct classes. Following the current trends in this field, we decided to deploy it in the cloud. Due to some factors, AWS was chosen in this case.

2 Brief description of a system

System is in-depth described in the Bachelor Thesis [1]. Its code is publicly available at GitHub.

In short, it has been built to be easily scalable, since running many virtual machines consumes a lot of resources. In the center of the system there is a main app, which is responsible for managing and distributing labs across workers, exposes API for frontend and saves information in database. The next part of the system is worker. It is meant to be started on each physical host and is responsible for managing container/VMs on a host and exposes an API for the main app to control labs over REST interface.

Diagram 1 might help with understanding this architecture.

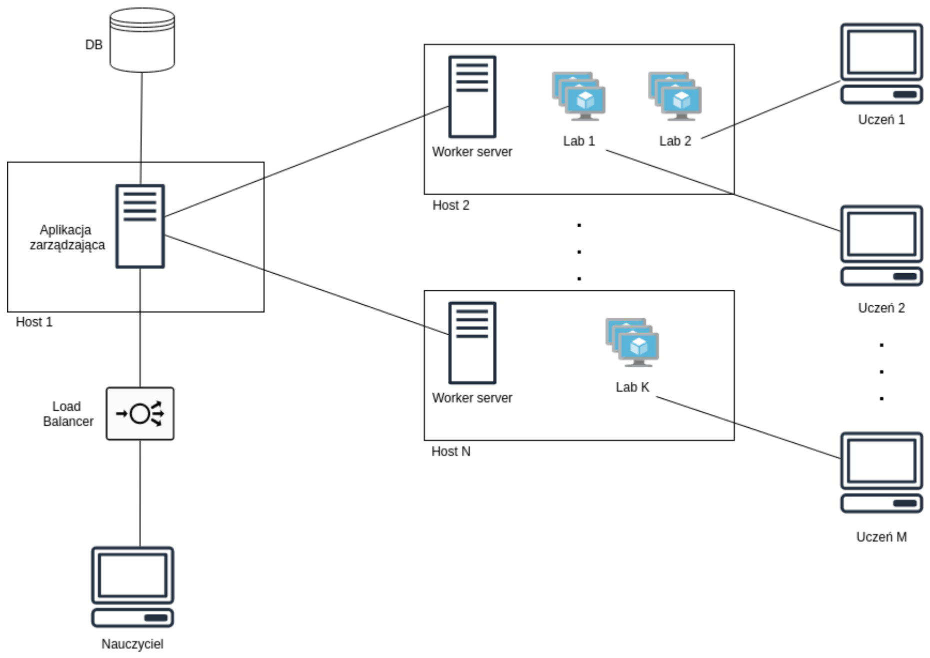

Figure 1: System architecture

2.1 System deployment preparation

All modules of the system were already contenerized. There was also a CI/CD setup for this project, which published all containers to the Azure Container Registry, so it was easy to download and use them on AWS.

The only change that was needed in the application code was a database migration from SQLite, which was used, to Postgres. Further change in this area made by us was adding the SQLAlchemy library which is used to support Postgres and is crucial when using system with AWS RDS.

3 Cloud architecture

From the very begging we decided to use Terraform [8] [14] [9] [11] to make architecture automated and to cut costs by destroying infrastructure during development. The deployment code is available at GitHub.

Because the app was contenerized the natural choice was to use container orchestrator for it. AWS supports two options: Elastic Container Service (ECS), which is a custom implementation of an orchestrator and Elastic Kubernetes Service (EKS) which allows to create a Kubernetes cluster. We decided to use ECS service.

ECS creates set of EC2 machines and connects them into a cluster and then allows to deploy containers there. It is supporting both simple and advanced container options like ‘priviledged‘, which was necessary in our case since inside a worker container VirtualBox was needed.

Final infrastructure is presented on the diagram 2.

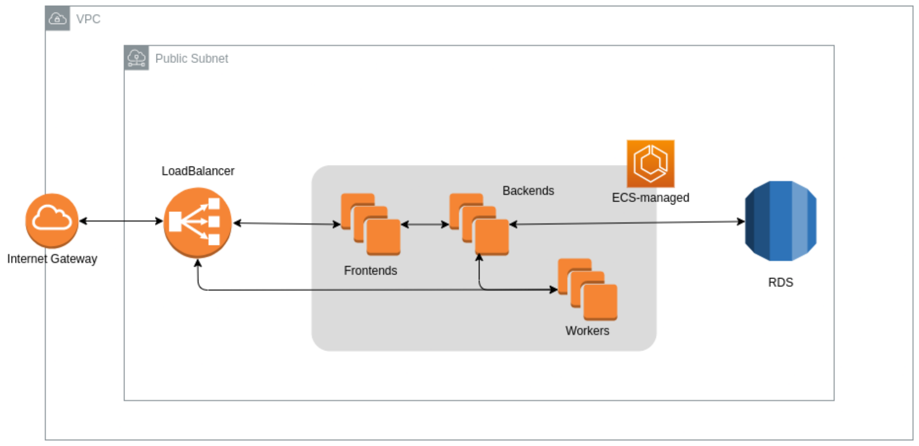

Figure 2: Cloud architecture

The system is placed inside VPC and then inside one public subnet. Also database is there due to the fact that we needed to populate it with external script. It is not a good pattern and generally databases along with backends should be placed inside separate privete subnets. For database we have used Amazon RDS with postgres provider.

ECS [6] [12] [10] operates on an EC2 cluster which is created and managed using autoscaling group. Cluster is made of three job definitions, one for each module: frontend, backend and worker. In an usual deployment it is enough to have one frontend and backend instance on the same EC2 and N workers each of which are placed on one, separate EC2. Mentioned default placement constraints [3] are also included in the Terraform code. Containers are pulled from Azure Container registry. Secrets for this registry are managed using Amazon Secrets Manager [4]. Also a custom AMI was created, which is described in the next subsection.

On top of that there is also a load balancer setup. It is connected to the default Internet Gateway and routes traffic to the frontend by default and to backend if it begins with ‘/api‘. It has also a possibility to support many frontend or backend instances.

3.1 Custom ECS image

Some of our labs depends on VirtualBox as a provider. P4 lab that we aimed to deploy may serve as an example. When Terraform is used in docker it needs to be also installed on the host system, since container does not support virtualization and needs to operate on a host supervisor.

ECS instances have to have also a special image that contains a program which manages containers on an EC2 instance. For almost all Linux distros there is already created an AMI that is marked as "ECS-optimized". However, none of them has VirtualBox installed. We decided to create custom AMI that contains both elements.

Tool that is commonly used for creating such images is Packer [7].It allows to use virtual machines from many sources as a base, install more tools and create another virtual machine, even for different provider. Using this tool and articles [5] [2] we were able to create an AMI which was later used as an EC2 in ECS cluster.

4 Problems

Described architecture allowed us to successfully run the app in the cloud. However the problem that we encountered when trying to run a lab using the system was the fact that Amazon doesn't support nested virtualization. Therefore VirtualBox, despite installing successfully, wasn't able to run any nested VM. Labs that were based on docker worked pretty well. The only way to run them on Amazon is to choose bare-metal instances, which are very expensive.

To overcome this issue we considered three possible solutions. First approach was to abandon automatization in the system and create AMI from lab machine. Second was to migrate lab to docker and not use VirtualBox. The last one was to use Azure for workers since it supports nested virtualization.

5 Creating Amazon AMIs

Due to tight time constraints we decided to follow the first way. Again using Packer and Vagrantfile from P4 tutorials [13] we built an Amazon AMI that is a cloud equivalent of VirtualBox machine. The code is also available in the lined repository.

6 Evaluation

Using mentioned approach, we were able to run about 10 sessions of laboratories, each of which needed 6-7 instances of virtual machines. Students were able to connect to machines using RDP and do the assignments. In their opinion it worked smoothly and was a better solution than downloading tons of megabytes before labs.

One negative thing that we mentioned was that we needed to use large instances with 4 vCPUs and 7.5 GB of RAM. It is a lot for one instance but on smaller ones it was not so smooth. Probably AMI was not enough optimized. All of those labs used about 50$. Development phase of the application deployment took abot 10$.

7 Lesson learnt

The biggest outcome of a project is that the system should be designed a bit differently from the very beginning. Current worker was not optimized for a cloud and did not work without nested virtualization. As a solution for above mentioned issue there should be added an additional type of worker that works more like a proxy between the system and a cloud provider and that should run machines from images optimized for a specific cloud. For each cloud separate worker should be added.

Besides this failure we were able to successfully run laboratories and deploy the application to the cloud (only for docker labs) which is also a great success. Everything is also fully automated so it is easy to use and modify without generating too big costs.

References

[1] Arkadiusz Kraus and Mateusz Naróg. “Automated creation of virtual lab on programmable network”. In: (2021).

[2] AWS. *AWS ecs variables.* https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-agent-config.htmlecs-instance-attributes.

[3] AWS. *AWS task placement.* https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-placement-constraints.html.

[4] AWS. *AWS task private auth.* https://docs.aws.amazon.com/AmazonECS/latest/developerguide/private-auth.html.

[5] AWS. *Installing ECS agent.* https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-agent-install.html.

[6] Ulysse Carion. *Setting up ECS with Terraform.* https://blog.ulysse.io/post/setting-up-ecs-with-terraform/.

[7] Hashicorp. *Packer.* https://www.packer.io/.

[8] Hashicorp. *Terraform.* https://www.terraform.io/.

[9] Hashicorp. *Terraform AWS docs.* https://registry.terraform.io/providers/hashicorp/aws/latest.

[10] Sean Hull. *How to setup an Amazon ECS cluster with Terraform.* https://medium.com/@hullsean/how-to-setup-an-amazon-ecs-cluster-with-terraform-87952988f239.

[11] Sam Meech-Ward. *Terraform — VPC, Subnets, EC2, and more.* https://sammeechward.com/terraform-vpc-subnets-ec2-and-more/.

[12] Tim Okito. *Get started with AWS ECS using Terraform.* https://medium.com/warp9/get-started-with-aws-ecs-cluster-using-terraform-cfba531f7748.

[13] P4. *P4 Virtual Machine.* https://github.com/p4lang/tutorials/tree/master/vm.

[14] Rolf Streefkerk. *How to setup a basic VPC with EC2 and RDS using Terraform.* https://dev.to/rolfstreefkerk/how-to-setup-a-basic-vpc-with-ec2-and-rds-using-terraform-3jij.