**Deploying to IaaS (AWS ECS)**

**Introduction to AWS**

* AWC (Amazon Web Services) ECS (Elastic Container Service)
* We will upload our Docker image to ECR (Elastic Container Registry)
* IaaS:
  + Cons:
    - More time in configuration
  + Pros:
    - More freedom to configurate
    - More scalable
    - More cost effective if you know what you’re doing with your configuration

**AWS Costs and Caution**

* Minimal cost seems to be 5 dollars a month

**Intro to AWS ECS**

* Task definition: Any instances of containers running together. It includes information about:
  + The Docker image that you want to use with each container in your task
  + How much memory and CPU to use with each task
  + The logging configuration
  + If a task should continue to run if the container finishes or fails
  + The command the container should run when it is started
* Service: Allows you to run and maintain multiple instances of a task definition
* Cluster: Parent grouping of tasks and/or services
  + Fargate: More expensive, less to manage and configure. Allows you to run containerized applications without the need to provision and manage the backend infrastructure. You can either:
    - Put multiple containers into the same task definition, or
    - Deploying containers separately in multiple task definitions
  + EC2: Less expensive, more to manage and configure. Allows you to run your containerized applications on a cluster of Amazon EC2 instances that you manage

**Container Orchestration Options: Kubernetes, ECS, Docker Swarm**

* Container Orchestration is all about managing the life cycle of containers
* Kubernetes
  + Is self-managed, meaning that you can deploy it on premises, private clouds and public clouds
  + Complex installation
* ECS
  + Fully-managed by AWS
  + Simpler
* EKS
  + Mid 2018
  + Managed Kubernetes in AWS
  + Middle-ground between Kubernetes and ECS

**Create an AWS Account**

**Setting Permissions with IAM (Identity and Access Management)**

* A group (collection of permissions) was created. Permissions added:
  + Amazon EC2 Container Service Full Access
  + Amazon EC2 Container Registry Full Access
  + IAM Read Only Access
* A user was created
  + This user will have the group of permissions that we’ve just created
  + We generated an Access key ID for this user and a Secret access key

**Installing the AWS CLI**

* We installed the AWS CLI to setup our ECS

**Configuring the AWS CLI**

* We configured AWS CLI by providing the user Access key ID and its Secret access key

**Intro to Elastic Container Registry (ECR)**

* Used to store Docker images where other AWS services will able to access them
* You find it on AWS
* We created a repository called udemy-ml-api, which comes with a URI

**Uploading Images to the Elastic Container Registry (ECR)**

* We will build an image locally and push it to ECR

**Creating the ECS Cluster with Fargate Lauch Method**

* We’ll set the container definition, task definition, service and cluster
* We copy the latest image from our ECR
* 1 We define a container definition using the image
  + We set the port to be 5000 because that’s what we have defined in Gunicorn and in our Docker file
  + We set the PIP\_EXTRA\_INDEX\_URL which is our Gemfury URL
* 2 After creating the container definition we end up also creating the task definition
* 3 Default service
* 4 Default cluster, the name of our cluster is ml-api-cluster
* Then the cluster is created
* After our task starts running we will be able to use the DNS of the Fargate container to check that our api is running
* Then, grabbing the Task Network Public IP and append the port 5000 and type /health or /version, we get an answer back

**Creating the ECS Cluster with the EC2 Lauch Method**

* When using EC2 we have to create our own task definition
* The information needed to create it was:
  + Name: ml-api-task-definition-revised
  + Task Role: ecsTaskExecutionRole (default), it makes sure that we have the necessary permissions
  + Network Mode: Bridge, allows us to map our host to our container port
  + Task execution role: ecsTaskExecutionRole
  + Task memory: 128 MiB
  + Task CPU: 512 units
  + Container Definition (same as in the Fargate method)
    - But now, we have the Host port, which we defined to be 80
* Creating the cluster by selecting the EC2 Linux + Networking
  + Name: ml-api-updated
  + EC2 instance type: t2.micro
  + Number of instances: 1 (you can scale the number of instances depending on your traffic needs)
* After the cluster is created, we ran a new task using the EC2 launch type
  + The task definition that we used was the one that we have created just a moment ago (ml-api-task-definition-revised)
* After the task starts running, we grab the External Link to access our api endpoints. We don’t need the port mapping anymore because we have done the host container mapping

**Updating the Cluster Containers**

* Deploying a new image and updating our containers in our cluster to make sure that they are running the latest image
* The api version file was updated from 0.2.2 to 0.2.3
* Then a new image was built, tagged and pushed up to our AWS ECR
* To update the task we need to run:
* To get a consistent URL we need to use an application load balancer

**Tearing down the ECS Cluster**

**Deploying to ECS via the CI Pipeline**

* We added a new job to our CircleCI file which:
  + Installs the AWS CLI
  + We log in:
    - Eval takes the output and run that
    - We update our cluster, using environment variables from CircleCI
      * We need to add the following environment variables to CircleCI:
        + AWS\_ACCESS\_KEY\_ID
        + AWS\_ACCOUNT\_ID
        + AWS\_DEFAULT\_REGION
        + AWS\_SECRET\_ACCESS\_KEY
* Makefile commands:
  + Building the image
  + Tagging the image
  + Pushing the image
* Tweaks to the production config

**Wrap Up**