**Chapter 11 to 15 is important**

**DOCKER**

* Docker is a container technology. A tool for creating and managing containers
* Container: a package of code and dependencies to run that code(e.g. python code + runtime)
* Same container always yields the exact same application and behaviour
* Use case
  + 1 team member is working on advanced python and other is working on lower version of python, they work on interface and want to test something
* Provides us with same dev/prod environment
* Virtual machines: In virtual m/c we have operating system running inside every virtual m/c on top of our base operating, this consumes lot of space and memory.
  + Pro
    - Separate env
    - Env specific config is possible
  + Con
    - Redundant duplication, waste of space
    - Performance can be slow, boot time can be long
* Containers: here we have docker running on our base operating system and we have container support os in between them. On top of docker engine we have container containing our app and dependent libraries
  + Low impact on OS, very fast, minimal disk space usage
  + Sharing, rebuilt is easy
  + Encapsulate app/env instead of whole machine
* Docker file – define how container should be setup
* From node:14
  + The FROM command in a Docker file is one of the most crucial instructions, as it specifies the base image from which your Docker image will be built. The base image is the starting point for your image and contains the essential files and dependencies needed for your application or environment.
* Workdir /app
  + Sets the Current Directory: WORKDIR sets the directory inside the container where commands will be executed. If the directory doesn’t exist, it will be created automatically.
  + Applies to Subsequent Instructions: Once set, all subsequent instructions in the Docker file will use this directory as their current working directory.
* COPY Package. Json .
  + Copy file from host machine to image path
* RUN npm install
  + Install dependencies
* COPY . .
  + Copy the rest of the application code into the container
* EXPOSE 3000
  + It indicates that the application running inside the container will listen on port 3000. This is common in Node.js applications, as they often run on port 3000 by default.
  + While EXPOSE doesn't actually open the port for external access, it serves as documentation for others to understand which ports the application is using.
  + To make the container's port accessible from the host machine or external networks, you need to map the container's port to a port on the host machine using the -p or -P options when running the container with docker run.
  + docker run -p 3000:3000 my-image
    - This command maps port 3000 of the host machine to port 3000 of the container, making the application accessible at http://localhost:3000 on the host.
  + CMD[python. ‘app.py’]
    - The CMD instruction in a Docker file is used to specify the default command that should be run when a container is started. Unlike other instructions, CMD does not execute when the image is built; instead, it sets a default command to be executed when the container runs.
    - CMD ["executable", "param1", "param2"]
  + Docker build .
  + Docker run -p 3000:3000 <imageid>
  + Docker ps : list all running containers
  + Docker stop <container name or id>
  + Images: template/blueprint for containers
  + Containers: running unit of software
  + Docker run -it <image name> : it will run interactive shell inside the container
  + Docker build .
  + The docker start command is used to start an existing, stopped container. It does not create a new container; it only starts a container that has been created previously (whether by docker run or docker create).
  + Docker stop
  + Images are read only once created we cannot change them; you need to rebuild after making changes
  + Images are layer based, how Docker Layers Work
    - 1. Layer Structure
      * Read-Only Layers: Each Docker image is composed of a series of read-only layers stacked on top of each other. Each layer represents a set of changes or instructions that are applied sequentially to build up the final image.
      * Base Image: The bottom layer of an image is usually a base image, like alpine, ubuntu, or node, which contains the operating system and essential packages.
      * Intermediate Layers: Every subsequent instruction in the Docker file (such as RUN, COPY, ADD, etc.) creates a new intermediate layer. These layers contain the changes made by that specific instruction.
      * Final Image: The final Docker image is the sum of all these layers combined. When you pull or build an image, Docker assembles it by stacking all these layers together.
    - 2. Layer Caching
      * Build Efficiency: Docker caches layers to speed up the build process. If a layer hasn’t changed between builds, Docker reuses it instead of rebuilding it from scratch. This can significantly reduce build times, especially in large projects.
      * Example:
      * If you have a Docker file with multiple RUN commands, and you only change one of the commands, Docker will only rebuild the layers from that point onward, reusing the previous layers that haven’t changed.
      * Layer Reuse: Since layers are reusable, multiple images can share the same base layers. For example, if two images are built from the same base image, they will share those base layers, saving storage space and reducing network bandwidth when pulling images.
* We should copy package.json, followed by npm install, followed by copy . . , to make sure that whenever we change code and then we rebuilt we end up saving time to execute npm install
* If we have 1 image and 2 containers, there will be just 1 source code and 1 environment in image and container will just refer that
* Docker run will create new container and by default run in attach mode i.e. We can see the logs in console, whereas docker start by default runs in detach mode
* We can use -d in docker run command to run in detach mode, -a for attach mode
* Docker logs <container name>
* docker attach CONTAINER attaches you to a running Container with an ID or name of CONTAINER.
* To provide input to code use -it in docker run
* You cannot remove a running container
* Docker rm <container name or id>
* Docker container prune: to remove all stopped container
* Docker rmi – to delete the images
* Docker run –rm : it will automatically delete the container on exit
* Docker image inspect <image-id> : provide all info about the image
* Docker command to copy file from local to running container
  + docker cp <source path> <container\_name\_or\_id>:<destination path>
* using –name in docker run we can give a name to our container
* image tags – name: tag
  + docker build -t my-app:v1.0 .
  + docker tag old-tag new-tag
* docker login – will login and allow us to push to docker hub
* when we push an image to docker hub it does not push the node image instead it relies on image that is already present in docker hub
* docker pull – to pull images from docker hub

**MANAGING DATA and WORKING WITH VOLUMES**

* when our code generates some temp data and if we write it to a file(which is OK if we lose it) this file is stored in container layer(read/write) and not in image. If we delete the container, we lose the data
* if we want to store permanently, we store it in volume
* Volumes are folder on host machine hard drive which are mounted into containers
* To add volume
  + VOLUME ["/path/in/container"]
    - VOLUME ["/app/data", "/app/logs"]
  + docker run -v /host/data:/app/data image-name
    - In this case, /host/data on your host will be mapped to /app/data in the container, overriding the volume set in the Docker file if there’s a conflict.
* docker volume ls : shows all volumes
* volumes are of 2 types:
  + anonymous volume – lost when container is stopped, created by docker on our system if we don’t specify any named volume
  + named volume
* volumes are managed by docker and we don’t know where on host machine it is created whereas in bind mount, we control the path, so we know where it is created. We can use this property of bind mount to do interactive development in container.
* To do mount volume
  + docker run -v {path to project}:/app image-name
  + make sure in docker preferences docker has access to the parent folder
* when we are doing bind mount inside the container’s app folder, the data in app path is getting overwritten, to overcome this we can create one more anonymous volume where the file from app/modules will be written temporarily
  + docker run -d -p 3000:80 –name feedback-app -v feedback:/app/feedback -v “{path to project}/app” -v /app/module feedback-node: volumes
  + even after the above step html changes are reflecting instantly but for node, we will add dependency in package.json
* anonymous volume : docker run -v /app/data
* named volume : docker run -v data:/app/data
* bind mount: docker run -v /path/to/code:/app/code
* bind mount read only(container can access folder in read only mode): docker run -v /path/to/code:/app/code:ro
* docker run -d -p 3000:80 –name feedback-app -v feedback:/app/feedback -v “{path to project}/app” -v /app/module -v /app/node\_modules feedback-node: volumes
* In production we don’t use bind mount
* what will happen if the mount point path already contains some existing files
  + When you use a bind mount, you directly map a directory from your host machine to a directory (mount point) in the container. The host directory completely replaces the contents of the container directory at the mount point, hiding any files that existed there previously.
  + When using a volume, Docker tries to manage things a bit differently
    - If the container directory already contains files and the volume is empty, Docker copies those files into the volume and they remain accessible.
    - If the volume already has data, the files inside the volume overwrite what is in the container’s directory, hiding any files that were already in the container's mount point.
* If a volume is mounted on a subfolder (e.g., /app/subfolder), it will **override** any content from the bind mount for that specific subfolder.
* Docker will ensure that the volume mount takes precedence over the bind mount for that particular subdirectory (/app/subfolder).
* We can create .dockerignore file to prevent files from being copied in image folder
* Argument
  + build time only
  + Used to pass variables during the image build process.
  + Available only during the image build process and **not** accessible once the container is running.
  + Arguments are defined in the Dockerfile and can be passed to docker build via the --build-arg flag.
  + ARG VERSION=latest
  + RUN echo "Building version $VERSION"
* Environment variable
  + Runtime.
  + Used to pass variables that the container’s processes can access while it is running.
  + Available inside the container during its lifetime and can be accessed by processes running inside the container.
  + Can be set in the Dockerfile, passed at runtime using the -e option, or stored in an env-file
  + ENV APP\_ENV=production
* ARG: Ideal for configuring how the image is built (e.g., installing specific versions of dependencies).
* ENV: Best for setting configuration options that the application needs while running (e.g., database connection strings, mode of operation).

**NETWORKS**

* Request from container to www – doesn’t require any change
* Request from container to host machine(local machine)
  + host.docker.internal is a special DNS name used in Docker to allow containers to connect to services running on the host machine (local system) from within a container. It is particularly useful when you need a container to communicate with services on the host, such as databases, web servers, or any other local network services.
  + Use case: Normally, containers are isolated from the host, and trying to connect to localhost or 127.0.0.1 inside the container will refer to the container's own internal network, not the host machine. host.docker.internal provides a way to refer to the host machine in a cross-platform manner, without needing to know its IP address.
  + Accessing a Database Running on the Host
    - Let's say you have a PostgreSQL database running on the host machine, and you want to connect to it from within a Docker container:
    - Run the database on the host, for example on localhost:5432.
    - Use host.docker.internal inside the container to refer to the host's IP address.
* Request from container to container
  + Inspect the container running mongo db., pick the ip from there and use it in connection URL from application – not a good solution
  + We can also create container networks
    - Create a network: docker network create my-network.
    - Run MongoDB on the network: docker run --name my-mongo --network my-network -d mongo.
    - Run your application on the same network with the MongoDB connection string: docker run --name my-app --network my-network -e MONGO\_URL="mongodb://my-mongo:27017/mydb" my-app-image
* Exposing port using -p is only required when we are connecting to container from local machine or from outside container network, so in above example we don’t use -p option while running mongo db. as container

**Building multi container application with docker(imp)**

* To start mongo db.:
  + Docker run –name mongodb –rm -d -p 27017:27017 mongo
  + Docker run –name goals-frontend –network goals-net –rm -p 3000:3000 -it goals-react

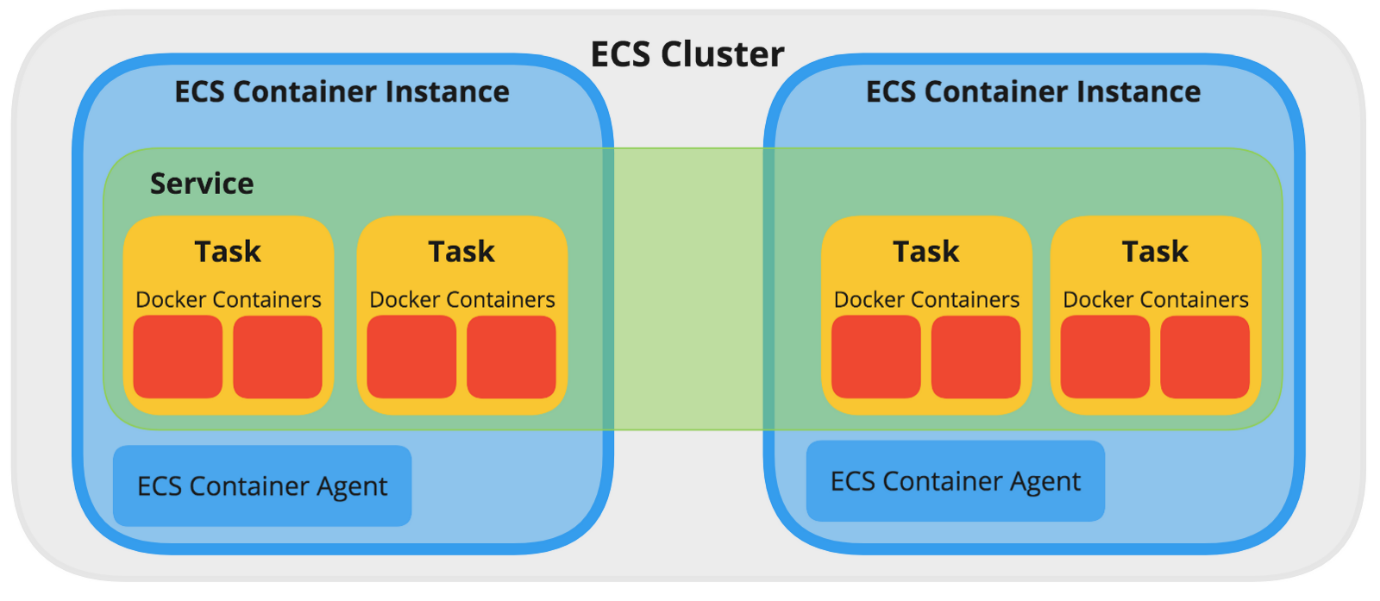
**DOCKER COMPOSE**

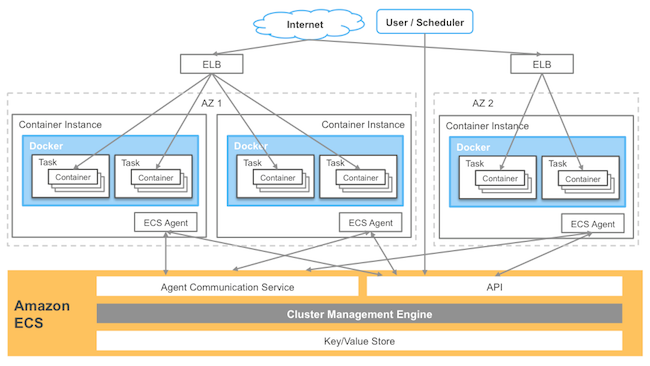
* Docker Compose is a tool used to define and manage multi-container Docker applications. It allows you to define all the services (containers), networks, and volumes your application needs in a single docker-compose.yml file, which simplifies managing complex applications with multiple containers.
* <https://docs.docker.com/compose/>
* Whatever we are specifying in the run command like image name, container name, port, network, volume, we can specify in a centralized place like compose file
* All the services defined in compose file will be placed in the same network by default, so we don’t need to specify network specifically
* Docker-compose up – to start all containers
* Docker-compose down – to bring down all containers, does not delete volume
* To also delete the volume, we use: docker-compose down -v
* Instead of we building the image, we can pass the instruction to docker compose to build
* Although the name assigned by docker compose to container is different we can use the name given in service of docker-compose to refer in code(URL of mongo db to make connection to db)

**Utility Container**

* Run the node container in interactive and detached mode
  + docker run -it -d node
* Get container id or name
  + docker ps
* Attach to running container. This command allows you to open a shell session inside the running container and interact with it.
  + docker exec -it <container\_id\_or\_name> <command>
* to override the default command when container starts
  + docker run <image\_name> node –version
* to create new project inside container and reflect in our local:
  + docker run -it -v /path/to/project:/app node-util npm init
* So, whatever we mention in CMD in docker file will get replaces by command we mention in docker run but whatever we mention in ENTRYPOINT will get prefixed to the command passed in docker run

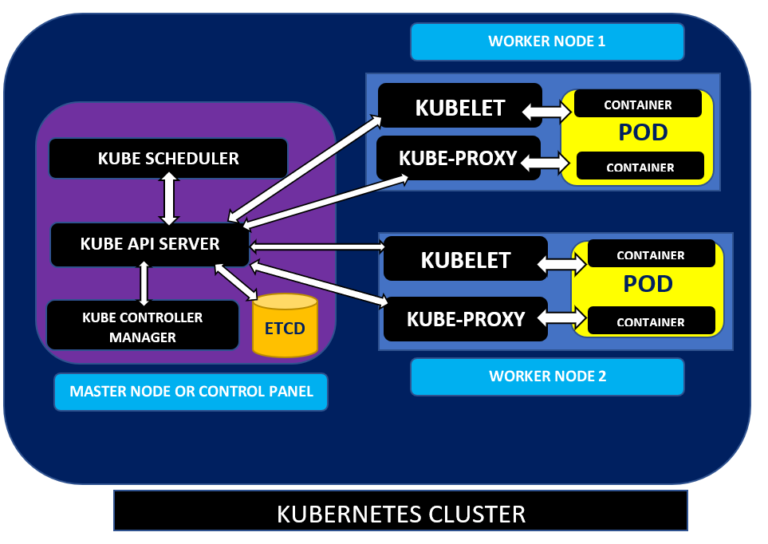
**Deploying docker container**





* It will build the image and give the tag as username/image-name:tag
  + docker build -t <your-dockerhub-username>/<image-name>:<tag> .
* AWS ECS - To run Docker containers on AWS ECS (Elastic Container Service), you typically create the following:
  + Task Definition: Defines the containers to be run, including their configurations (e.g., memory, CPU, image, networking, etc.).
  + Service: Manages and maintains the desired number of task instances running.
  + Cluster: A logical grouping of tasks or services that run your containers.
* Whatever we pass in docker run similar things we configure in console of aws ecs task definition
* To make ecs pull in new image from docker hub 🡪 task revision in ecs 🡪 update task
* Docker compose doesn’t provide with all options to deploy image in cloud
* The Docker networking model does not fully apply in the same way in AWS ECS (Elastic Container Service). When deploying Docker containers on ECS, especially when using the Fargate launch type, AWS manages much of the networking for you, and it follows a different model.
* In ECS, especially with Fargate or EC2 instances, networking is handled at a higher level by AWS, and many Docker-specific networking concepts don’t directly apply.
* To make two containers run in the same network in AWS ECS, you need to configure them to be part of the same ECS Task Definition. When two containers are defined within the same task definition, they will share the same Elastic Network Interface (ENI) and network settings if you are using the awsvpc network mode.
* Task – basically contains all of image info like network, volume, image name, port etc, image url, things that we provide in docker run. It defines 1 or more containers
* Service – contains details about alb, vpc, subnet, asg
* Cluster – logical grouping of service or tasks
* Start from 152

**KUBERNATES**

* Kubernetes (often abbreviated as K8s) is an open-source platform designed for automating the deployment, scaling, and management of containerized applications.
* ECS supports all things below
  + Containers might crash down and needs to be replaced(Health Check)
  + We might need more container on spike(ASG)
  + Incoming traffic should be distributed equally(Load balancer)
* ECS is specific to AWS, Kubernetes is generic
* Kubernetes is like docker compose but instead of multi container, it is for multiple machine
* Core concepts and architecture
  + Cluster: A set of nodes (machines) where Kubernetes runs containerized applications. Combination of master and worker node.
  + Pod(Container) – The smallest deployable unit in Kubernetes. A pod can contain one or more containers that share resources like networking and storage
  + Nodes: your machine/virtual instances. E.g. ec2
  + Worker Node: Executes the application containers
  + Proxy config: In Kubernetes, configuring a proxy is often necessary when you need to route external traffic into the cluster or when pods within the cluster need to access external resources through a proxy server. Typically handles external traffic going in or out of the cluster.
  + Kube proxy: Manages internal cluster service networking and traffic routing between services and pods. Handles internal traffic within the cluster.
  + Services: In Kubernetes, a **Service** is an abstraction that defines a logical set of Pods and a policy by which to access them. Services provide a stable network endpoint, even if the underlying Pods change or are rescheduled on different nodes. This is crucial because Pods in Kubernetes are ephemeral and can be created or destroyed dynamically. Services offer a consistent internal IP (ClusterIP) and DNS name for accessing the underlying Pods, regardless of their lifecycle (creation or deletion).
  + Master node: Controls the cluster, making decisions such as scheduling and scaling
  + Kubelet: communication between master and worker node
  + Kube API server: API for KUBELETs to communicate
  + Scheduler: watches for new pods, select worker nodes to run them
  + Kube controlled manager: watches and controls worker nodes, correct number of Pods and more
  + Cloud controlled manager: same as kube controlled manager but for cloud specific
  + Kubectl: a tool for sending instructions to the cluster
  + Minikube is a lightweight tool that allows you to run a Kubernetes cluster locally on your machine, typically for development or testing purposes. It creates a single-node Kubernetes cluster that you can use to try out Kubernetes features without the need for a full-blown production environment or cloud provider
* 
* Our responsibility
  + Create cluster and the node instances(worker/master node)
  + Setup API server, kubelet and other Kubernetes s/w on nodes
  + Create ALB, filesystem
* What Kubernetes will do:
  + Create Pods and manage them
  + Monitor Pods, recreate them, scale them
* Containers inside pod can communicate via localhost
* For pods to be managed for you, you need a controller(e.g. a deployment)
* In Kubernetes, a **Deployment** is a controller that manages the creation and management of **Pods** using **ReplicaSets**. It is a higher-level abstraction designed to make it easier to handle application updates, scaling, and self-healing. With a Deployment, you can declare the desired state of your application (e.g., which container image to run, how many replicas, etc.), and Kubernetes will ensure that your application's current state matches the desired state.
* Command to start minikube:
  + minikube start --driver=docker
  + minikube start --cpus=4 --memory=8192
  + minikube status
* Command to create deployment
  + kubectl create deployment <deployment-name> --image=<image-name>
  + kubectl create deployment nginx-deployment --image=nginx --replicas=3
  + kubectl expose deployment nginx-deployment --port=80 --target-port=80
  + kubectl create deployment myapp-deployment --image=username/my-app:latest
  + kubectl get deployments
  + kubectl get pods
  + kubectl delete deployment <deployment-name>
* kubectl create deployment <deployment-name> --image=<image-name>, this will send command to master node where scheduler analyses the currently running pods and find the best node for the new pods🡪 kubelet manages the pods and container
* Service object
  + exposes pods to the cluster or externally
  + pods have internal ip by default, it changes when a pod is replaced
  + finding pods is tough as ip changes all the time
  + services groups pods with a shared ip
  + services can allow external access to pods
  + reaching a pod from outside the cluster is not at all possible without services
* Command to create service:
  + kubectl expose deployment <deployment-name> --type=<service-type> --port=<port> --target-port=<target-port>
  + accessible from only the cluster: kubectl expose deployment nginx-deployment --type=ClusterIP --port=80 --target-port=80
  + For external access: kubectl expose deployment nginx-deployment --type=NodePort --port=80 --target-port=80
* To scale: kubectl scale deployment nginx-deployment --replicas=5
* Command to update new image in existing deployment(new images will be pulled only if they have a new tag):
  + kubectl set image deployment/<deployment-name> <container-name>=<new-image>:<tag>
* To check the rollout status of the update:
  + kubectl rollout status deployment/myapp-deployment
* To view the history of rollouts (versions) for the deployment:
  + kubectl rollout history deployment/myapp-deployment
* To roll back to a previous version if something goes wrong
  + kubectl rollout undo deployment/myapp-deployment
* command to rollback to a specific deployment
  + kubectl rollout undo deployment/<deployment-name> --to-revision=<revision-number>
* Imperative
  + providing step by step command to achieve the desired result in Kubernetes
  + comparable to docker run
* Declarative – You describe what you want the system to look like, and the system determines how to achieve that state.
  + In Kubernetes, a **resource definition** is a way to describe the desired state and configuration of various objects within a cluster. These definitions are usually written in YAML or JSON format and are used to manage and configure different types of resources like Pods, Deployments, Services, ConfigMaps, and more.
  + Comparable to docker compose
  + Selector field in deployment yaml
    - The selector field specifies the criteria used to identify which Pods are managed by the Deployment. This is important because it allows Kubernetes to distinguish between Pods created by this Deployment and Pods created by other sources (e.g., other Deployments, manually created Pods).
    - The selector helps ensure that only the Pods matching the specified labels are considered part of this Deployment. This is crucial for ensuring that the Deployment correctly handles the Pods it is responsible for, especially during scaling or updates.
  + Selector in service.yaml is slightly different from deployment.yaml
  + Command to get the url to hit in browser:
    - Minikube service <deployment name>

**Kubernetes deployment**

* Aws eks
  + Managed service for Kubernetes deployment
  + No aws specific syntax or philosophy required
  + Use standard Kubernetes configuration and resources
* Aws ecs
  + Managed service for container deployment
  + Aws specific syntax and philosophy applies
* In AWS EKS (Elastic Kubernetes Service), a **Cluster Service Role** is an IAM (Identity and Access Management) role that grants AWS services the necessary permissions to manage the underlying AWS resources for your EKS cluster. This role is associated with your EKS cluster and is used by the Kubernetes control plane(master node) to interact with AWS resources on behalf of the cluster.
* The **EKS Cluster Service Role** is an IAM role that EKS uses to provision and manage resources such as Elastic Load Balancers (ELBs), EC2 instances, networking components (VPC, subnets), and more.
* A **Node Group** in Amazon EKS (Elastic Kubernetes Service) refers to a group of Amazon EC2 instances (or nodes) that are part of your EKS cluster and run your containerized applications. These nodes are the machines where Kubernetes schedules your workloads, such as Pods.