Of course! Let's transform your lab notes into a super detailed and easy-to-understand guide. We'll break down every step, explain the "why" behind it, and answer all your questions in simple language.

Your Ultimate Guide to the Weather Forecast Application Lab

Imagine we're building a small online weather website. This website isn't just one big program; it's made of four smaller, specialized programs (microservices) that work together:

1. **Frontend:** The website you actually see and click on in your browser.
2. **Advertisement:** A service that decides which ads to show you.
3. **Forecast:** The brain that calculates the weather prediction.
4. **Recommendation:** A service that suggests what you should wear or do based on the weather (e.g., "carry an umbrella").

To manage these programs easily, we package each one into a **Docker Image** (like a sealed box containing the program and everything it needs to run). We then run these images as **Docker Containers** (an active, running instance of the image).

Phase 1: Preparing Our Tools (Docker)

Purpose: We need a tool to build and run our software "boxes" (containers). Docker is that tool. We install it on our cloud server (ECS).

- `yum install -y docker` : This is like using an app store (`yum`) to download and install the Docker software onto our server.
- `systemctl start docker` : This command starts the Docker “engine,” so it’s running and ready to use.
- `systemctl enable docker` : This tells the server to automatically start Docker every time the server reboots, so we don’t have to manually start it again.
- `docker version` : This is a quick check to see if we installed Docker correctly. It’s like opening an app to make sure it doesn’t crash on startup.

Phase 2: Building Our Software “Boxes” (Docker Images)

Purpose: We need to create the Docker images for our microservices so we can run them anywhere.

A) Building the Forecast Image from Source Code

- `cd /root` : We navigate to the main folder (`root`) on our server.
- `wget weatherforcast-master.tar` : We download a compressed file from the internet that contains the source code for the forecast service. Think of it as downloading a `.zip` file.
- `tar -xvf file` : We “unzip” the downloaded file to access the code inside.
- `cd into the file till you find dockerfile` : We navigate into the unzipped folder. The `Dockerfile` is a set of instructions that tells Docker how to build the image (what base system to use, how to copy the code in, etc.).

- `docker build -t forecast:v1 .` : This is the magic command. It reads the `Dockerfile` and builds a new Docker image, naming it `forecast:v1` (version 1). We do this again to create a `v2` , which might have improvements or bug fixes.

B) Loading Pre-Built Images

- `docker load -i frontend.tar` : This command loads a pre-built Docker image from a file on your computer into Docker's storage. We do this for `frontend` , `advertisement` , and `recommendation` because someone else already built these images for us and saved them as `.tar` files.

Q) What is the difference between `docker load` and `docker build` ?

A) This is a great question! Let's use a cake analogy.

- `docker build` is like **baking a cake from scratch**. You start with raw ingredients (your source code and a `Dockerfile` recipe) and you create a brand new cake (a new Docker image).
- `docker load` is like **buying a pre-baked, frozen cake from a store**. The cake (Docker image) was already baked (built) by someone else. You just need to take it out of its box (the `.tar` file) and put it in your freezer (Docker's storage) so you can eat (run) it later.

In the lab, we **built** the `forecast` image from source code, but we **loaded** the others because we downloaded them as ready-to-use packages.

Phase 3: Storing Our “Boxes” in a Warehouse (SWR)

Purpose: We can’t run our application from just one server. We need a central place to store our Docker images so that other servers can pull and run them. SWR (Simple Workload Repository) is Huawei Cloud’s warehouse for Docker images.

- **Create an organization:** An organization in SWR is like a company name or a brand. It helps group all your images together (e.g., `clouddemotest000`).
- **Upload images:**
  - **Generate temp login command:** This gives us a temporary password to prove to SWR that we are allowed to upload images.
  - `docker tag forecast:v1 swr.ap-southeast-3.../forecast:v1`: This command doesn’t move the image; it just gives it a new name tag that includes its “shipping address” (the SWR repository URL).
  - `docker push ...`: This is the actual “shipping” command. It uploads our tagged image from our local machine to the SWR warehouse.
- We then tag and push all the other images (`frontend`, `advertisement`, `recommendation`) in the same way.

Phase 4: Creating a “Computer Farm” (CCE Cluster)

Purpose: We need a powerful and reliable place to run our containers. A CCE (Cloud Container Engine) cluster is like a

farm of computers (called Nodes) that are managed as one big computer, perfect for running containerized applications.

- We create a cluster with **3 master nodes**. These are the “manager” computers that don’t run our app but instead manage and orchestrate the whole farm, making sure everything runs smoothly.
- We set up a **VPC network** with a **CIDR block of 172.16.0.0/16**. This is like building a private, secure road network inside our farm so that all our computers can talk to each other safely without the outside world interfering.
    - **Note on IP Ranges:** The different IP ranges (10.x.x.x, 172.16.x.x, 192.168.x.x) are just standard private address spaces. Using 172.16.x.x for a medium-sized setup is a common convention.

Phase 5: Adding “Worker” Computers (Worker Nodes)

Purpose: The master nodes are the managers, but we need “worker” computers to actually do the job of running our weather application containers.

- We buy **2 worker nodes** with `c6s.2xlarge.2` specs. These are the powerful computers that will host our `frontend`, `advertisement`, `forecast`, and `recommendation` containers.

Phase 6: Deploying Our Application

Now we tell our “computer farm” (CCE cluster) how to run our application.

A) Creating MySQL StatefulSet (The Database)

Purpose: Our application needs a database to store information, like user data or ad content. A database needs to remember its data even if it restarts, so we use a **StatefulSet**, which is perfect for stateful applications like databases.

- **Q) Why do we go to ConfigMaps? What will this step do?**
    - **A)** We go to ConfigMaps and Secrets first because our database needs sensitive information to start up, like a password. We don't want to write the password directly in our instructions because that's insecure. Instead, we store it in a special, secure locker called a **Secret**. The step of creating a secret is like writing the database password on a piece of paper and locking it in a safe. The StatefulSet will then be given the key to that safe when it starts.
- **Creating the StatefulSet:**
    - We select the standard `mysql` image.
    - We add the password as an **environment variable** from the secret we just created. This is like the container looking inside the safe to read the password when it starts.
    - We add **data storage** (a PVC - Persistent Volume Claim). This is like attaching an external hard drive to the database container. Even if the container is deleted and recreated, the data on this "hard drive" is safe and can be reattached.

B) Creating the Frontend, Recommendation, and Forecast Deployments

Purpose: These are the core services of our app. We use a **Deployment** because these are stateless services—if one crashes, we can just start a new one without worrying about losing data.

- We create a Deployment for each, selecting the image we pushed to SWR.
- For the **frontend**, we add a **LoadBalancer** service. This is like hiring a receptionist for a big office building. When someone from the internet wants to visit our website, the LoadBalancer (receptionist) directs them to one of the available frontend containers. We note the IP and port the receptionist is using so we can visit the site later.

Of course! Let me explain this in a much simpler way with a perfect analogy.

Think of the Advertisement Service as a BANK TELLER

Why the Bank Teller is Special

Imagine our weather app is a **bank**:

- **Frontend** = The bank's front door and welcome desk
- **Forecast** = A consultant who gives financial advice
- **Advertisement** = The **bank teller** who handles money

The bank teller is special because ONLY they need access to the VAULT (database)!

Q1: ConfigMap vs Secret - What's the difference?

ConfigMap = The bank's **ADDRESS and HOURS** (public information)

- `advertisement_database_host` = "123 Main Street, Vault Room"
- `advertisement_database_user` = "Head Teller"
- This is NOT secret - anyone can know this

Secret = The **VAULT COMBINATION** (super secret!)

- `advertisement_database_passwd` = "15-32-27"
- This MUST be kept private and secure

We separate them because:

- Directions (ConfigMap) can be public
- Passwords (Secret) must be private

Q2: Why are we even doing this?

Imagine if the vault combination was **painted on the bank wall**! If we needed to change the combination, we'd have to **rebuild the entire bank**!

Instead, we:

1. Keep the combination in a **secret envelope** (Secret)
2. Keep the bank address on a **changeable sign** (ConfigMap)

This is brilliant because:

- If we move the vault, we just **change the sign** (update ConfigMap)
- If we change the combination, we just **get a new envelope** (update Secret)
- We **never have to rebuild the bank!**

Q3: Why ClusterIP service?

The Advertisement service (bank teller) should **ONLY talk to other bank employees** - not random people on the street!

ClusterIP = An **internal bank intercom** system

- Frontend (welcome desk) can call: "Hey Teller, I need $500 for a customer!"
- Advertisement (teller) responds through the intercom
- **People outside the bank CANNOT use this intercom**

This keeps the teller safe and secure inside the bank.

The 3 Simple Steps:

1. **Write the directions** (ConfigMap)
    - "Vault is in Room 5B"
    - "Head Teller is on duty"
2. **Seal the combination** (Secret)
    - "Vault code: 15-32-27" 🔒

3. **Give the teller an intercom** (ClusterIP)
    - So only bank staff can talk to them

Why Only Advertisement Gets This Treatment?

Because **ONLY the bank teller needs the vault**!

- Forecast consultant = just gives advice (no vault access)
- Front door attendant = just greets people (no vault access)
- Bank teller = ONLY ONE who handles money (needs vault access)

The advertisement service is special because it's the ONLY one that stores and retrieves important data from the database!

Phase 7: Seeing the Result!

- **Go to the frontend workload -> access mode -> copy the public IP.**
- **Paste it in your browser along with the port number.**
- **Purpose:** This is the moment of truth! You are using the public IP of the LoadBalancer (the receptionist) to access the website you just built. The frontend service will then internally call the forecast, recommendation, and advertisement services to build the complete webpage you see.

Summary

In this lab, you:

1. **Built and packaged** your application into portable Docker images.
2. **Stored** them in a central warehouse (SWR).
3. **Set up a managed farm** of computers (CCE Cluster) to run them.
4. **Deployed your application** by giving the cluster instructions, carefully using Secrets for sensitive data and ConfigMaps for settings.
5. **Connected your services** internally (ClusterIP) and exposed the frontend to the world (LoadBalancer).
6. **Accessed your final, working weather forecast application!** Well done