Containers with Docker, ECS, and EKS

What is Docker?

Docker is a software development platform designed to deploy applications using container technology. Containers allow applications to be packaged in a standardized format, ensuring they run consistently across various environments.

Key Features of Docker:

- **Standardization**: Containers ensure predictable behavior across different operating systems.
- **Compatibility**: Docker works with any language, OS, or technology.
- **Use Cases**: Ideal for microservice architectures, migrating apps from on-premises to the cloud, and running any containerized application.

How Docker Works

Docker operates on an operating system by running a Docker agent on a server (such as EC2 instances). This agent can then start multiple Docker containers, each running different applications (e.g., Java, Node.js) or services (e.g., MySQL). Multiple instances of the same container can run simultaneously.

Docker Repositories

Docker images are stored in repositories:

- **Docker Hub**: A public repository with base images for many technologies and operating systems.
- **Amazon ECR (Elastic Container Registry)**: A private repository service by AWS, also offering a public gallery for image storage.

Docker vs. Virtual Machines

Docker containers share resources with the host system, making them lightweight compared to virtual machines (VMs), which have separate guest operating systems. While Docker containers can share networking and data, they provide less isolation than VMs. This resource efficiency allows more containers to run on a single server.

Architecture Comparison:

- **VM**: Infrastructure → Host OS → Hypervisor → Guest OS → Applications.
- **Docker**: Infrastructure → Host OS → Docker Daemon → Containers.

Getting Started with Docker

1. **Dockerfile**: Define how the Docker container will look.
2. **Build**: Create a Docker image from the Dockerfile.
3. **Push**: Store the Docker image in a repository (Docker Hub or Amazon ECR).
4. **Pull and Run**: Retrieve the Docker image from the repository and run it as a container.

Docker Container Management on AWS

Amazon ECS (Elastic Container Service):

- AWS's own Docker management platform.
- Facilitates running and managing Docker containers.

Amazon EKS (Elastic Kubernetes Service):

- Managed Kubernetes service by AWS.
- Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

AWS Fargate:

- Serverless container platform by AWS.
- Works with both ECS and EKS.
- Eliminates the need to manage underlying servers.

Amazon ECR:

- Used for storing Docker container images.
- Integrates with ECS and EKS for seamless container deployment.

Summary

Docker is a versatile and powerful container technology that simplifies app deployment and management. AWS offers several services to manage Docker containers effectively:

- **ECS** for Amazon's Docker management.
- **EKS** for Kubernetes management.
- **Fargate** for serverless container execution.
- **ECR** for storing container images.

Amazon ECS Overview

EC2 Launch Type

ECS (Elastic Container Service) allows you to run Docker containers on AWS. When using the EC2 launch type, you:

- Launch ECS tasks on an ECS cluster composed of EC2 instances.
- Provision and maintain the infrastructure yourself.
- Each EC2 instance must run the ECS agent, which registers the instance with the ECS service and specified ECS cluster.
- AWS starts or stops containers based on your ECS tasks, distributing them across the EC2 instances.

Fargate Launch Type

With the Fargate launch type:

- You run Docker containers on AWS without provisioning or managing EC2 instances.
- It's serverless; AWS manages the underlying infrastructure.
- You create task definitions to define your ECS tasks, specifying the CPU and RAM needed.
- AWS runs the ECS tasks, handling the infrastructure automatically.
- Scaling is straightforward—just increase the number of tasks.

Fargate is often preferred due to its simplicity and ease of management.

IAM Roles for ECS Tasks

EC2 Launch Type:

- **EC2 Instance Profile:** Used by the ECS agent to:
  - Make API calls to the ECS service to register the instance.
  - Send container logs to CloudWatch Logs.
  - Pull Docker images from ECR.
  - Access sensitive data in Secrets Manager or SSM Parameter Store.

ECS Task Roles:

- Applicable to both EC2 and Fargate launch types.
- Each ECS task can have a specific role allowing it to interact with different AWS services.
  - Task roles are defined in the task definition of your ECS service.

Load Balancer Integrations

To expose ECS tasks as HTTP/HTTPS endpoints, use:

- **Application Load Balancer (ALB):**
  - Recommended for most use cases.
  - Supports integration with Fargate.

- **Network Load Balancer (NLB):**
    - Recommended for very high throughput or high performance use cases.
    - Can be used with AWS PrivateLink.
- **Classic Load Balancer (CLB):**
    - Not recommended due to lack of advanced features and incompatibility with Fargate.

Data Persistence on Amazon ECS

To share data across ECS tasks, use data volumes, particularly Amazon EFS (Elastic File System):

- **Amazon EFS:**
    - Compatible with both EC2 and Fargate launch types.
    - Provides persistent, multi-AZ shared storage.
    - Serverless and pay-as-you-go.
    - Allows tasks running in any AZ linked to the EFS file system to share data.

Ultimate Combination:

- **Fargate:** For serverless task management.
- **Amazon EFS:** For serverless, persistent storage. Ideal for multi-AZ shared storage for containers.

Amazon ECS Cluster Creation and Configuration

Step-by-Step Guide

1. **Access ECS Console:**
    - Open the ECS console service.
    - Enable the new ECS experience on the top left.
2. **Create a Cluster:**
    - Go to "Clusters" and create a new cluster named `DemoCluster`.
    - Leave the default namespace as is.
3. **Select Infrastructure Options:**
    - **Fargate:** AWS runs containers for you on demand without managing compute resources.
    - **EC2 Instances:** You provide and manage the EC2 instances to run your containers.
    - **ECS Anywhere:** Allows running ECS containers on external instances, like your own data center.

   For this demo, enable both Fargate and Amazon EC2 instances.
4. **Configure EC2 Instances:**
    - Create a new auto-scaling group.
    - Choose the operating system (e.g., Amazon Linux 2 or Amazon Linux 2023).
    - Select `t2.micro` for the instance type (free tier eligible).
    - Set desired capacity: minimum 0, maximum 5.
    - No SSH key pair configured.
    - Leave root EBS volume size as is.
5. **Network Settings:**
    - Use the default VPC and the three available subnets.
    - Use an existing security group (default security group).
    - For auto-assign public IP, use the default subnet setting.
6. **Create the Cluster:**
    - Click "Create" to initiate the creation of the cluster.
    - While the cluster is being created, you can check the auto-scaling group settings.
7. **Verify Auto-Scaling Group:**
    - Go to "Auto Scaling Groups" on the left-hand side.
    - Verify the creation of the auto-scaling group named `Infra-ECS-Cluster`.
    - Desired capacity: 0, min capacity: 0, max capacity: 5.
    - The auto-scaling group spans three availability zones for better redundancy.

8. **Wait for Cluster Creation:**
    - Once the cluster creation is complete, explore the `DemoCluster`.
    - Verify that there are no services or tasks running initially.
9. **Explore Cluster Infrastructure:**
    - Check the "Infrastructure" tab within the `DemoCluster`.
    - Three capacity providers should be available:
        - **FARGATE:** For launching Fargate tasks.
        - **FARGATE_SPOT:** For launching Fargate tasks in spot mode (cost-saving).
        - **ASGProvider:** For launching tasks on EC2 instances through the auto-scaling group.
10. **Modify Desired Capacity:**
    - Adjust the desired capacity of the ASGProvider to 1.
    - An EC2 instance will be created and registered with the `DemoCluster`.
11. **Verify Instance Registration:**
    - Ensure the EC2 instance is in the running state and registered in the `DemoCluster`.
    - Confirm that the instance is available under "Container Instances" with resources available for task deployment.
12. **Ready for Task Deployment:**
    - The cluster is now ready to run ECS tasks using any of the available capacity providers (Fargate, Fargate Spot, EC2 instances).

This setup ensures that you can run and manage Docker containers on AWS using both serverless (Fargate) and managed (EC2) infrastructure options within a single ECS cluster.

Creating an ECS Service and Task Definition

Step-by-Step Guide

1. **Create a Task Definition:**
    - **Name:** `nginxdemos-hello`
    - **Docker Image:** `nginxdemos/hello` from Docker Hub.
    - **Infrastructure Requirements:**
        - Choose Fargate for serverless compute.
        - OS: Linux.
        - Task Size: 0.5 vCPU and 1 GB memory.
    - **Task Role:** Not specified for simplicity.
    - **Task Execution Role:** Default (automatically created if not present).
    - **Container Configuration:**
        - **Name:** `nginxdemos-hello`
        - **Image URL:** `nginxdemos/hello`
        - **Port Mappings:** Map port 80 of the container to port 80.
        - Leave other settings as default.
    - **Storage:** Fargate provides 21 GB ephemeral storage by default.
    - Create the task definition.
2. **Launch Task Definition as a Service:**
    - Go to Clusters and select `DemoCluster`.
    - Under Services, create a new service.
    - **Compute Option:**
        - Launch Type: Fargate.
        - Platform Version: Latest.
    - **Application Type:** Service (for long-running applications like web servers).
    - **Task Definition Family:** `nginxdemo-hello` (latest revision).
    - **Service Name:** `nginxdemos-hello`
    - **Service Type:** Replica with one task.
    - **Deployment Settings:** Leave as default.
3. **Configure Networking:**
    - Deploy in the default VPC across three subnets.
    - **Security Group:** Create a new one named `nginxdemos-hello`.
        - Allow HTTP traffic on port 80 from any source.
    - **Public IP:** Enabled.
4. **Load Balancer Configuration:**
    - Use an Application Load Balancer (ALB).
    - **Name:** `DemoALBForECS`
    - **Health Check Grace Period:** Default.

- **Port and Protocol:** HTTP on port 80.
- **Target Group:** `tg-nginxdemos-hello`
  - Protocol: HTTP.
  - Health Check Path: `/`

5. **Service Deployment:**
   - Confirm the deployment settings.
   - Deploy the service.
6. **Verify Deployment:**
   - Check the service to ensure one task is running.
   - Verify the target group is linked to the ALB.
   - Copy the DNS name of the load balancer and access it in a web browser to see the Nginx welcome page.
7. **Scaling the Service:**
   - **Increase Task Count:**
     - Update the service to have a desired number of tasks set to three.
     - Verify the new tasks are running and the ALB is distributing load among them.
   - **Decrease Task Count:**
     - Update the service to set the desired number of tasks to zero.
     - Ensure no containers are running to save on costs.
8. **Auto Scaling Group Adjustment:**
   - Adjust the auto-scaling group desired capacity to zero to ensure no EC2 instances are running.

By following these steps, you can create an ECS service using Fargate, deploy a containerized application, and manage its scaling and networking configurations.

ECS Service Auto Scaling Overview

Introduction to Auto Scaling

Amazon ECS Service Auto Scaling enables you to automatically increase or decrease the number of ECS tasks based on demand. This is done through AWS Application Auto Scaling, which supports three primary metrics for scaling:

1. **CPU Utilization** of the ECS Service
2. **Memory Utilization** (RAM) of the ECS Service
3. **ALB Request Count Per Target** from the Application Load Balancer (ALB)

Types of Auto Scaling

1. **Target Tracking Scaling:**
    - Automatically adjusts the number of tasks to maintain a specified metric target.
    - Useful for maintaining a consistent utilization level for CPU, memory, or request count.
2. **Step Scaling:**
    - Increases or decreases the number of tasks by a specified amount based on the breach of CloudWatch alarms.
3. **Scheduled Scaling:**
    - Adjusts the number of tasks based on a pre-defined schedule.
    - Ideal for predictable changes, such as traffic patterns during certain times of the day.

Fargate vs. EC2 Launch Type

- **Fargate:**
    - Fargate is a serverless compute engine for containers.
    - Makes service auto scaling simpler since there's no need to manage the underlying EC2 instances.
    - Ideal for those looking for a hassle-free scaling experience.
- **EC2 Launch Type:**
    - Requires managing the EC2 instances where the tasks run.
    - Scaling the service at the task level does not equate to scaling the EC2 instances in the cluster.

Scaling EC2 Instances in the Backend

When using the EC2 launch type, scaling the backend EC2 instances can be managed in two ways:

1. **Auto Scaling Group (ASG) Scaling:**

- Adjusts the number of EC2 instances in an ASG based on metrics such as CPU Utilization.
- Adds EC2 instances when CPU utilization increases.

2. **ECS Cluster Capacity Provider:**

   - An advanced feature that intelligently manages the capacity of the ECS cluster.
   - Automatically scales the ASG when additional capacity (CPU/RAM) is needed.
   - Preferred over manual ASG scaling due to its efficiency and automation.

Workflow of Auto Scaling

1. **Service with Initial Tasks:**

   - Example: Service A starts with two tasks.

2. **Increase in CPU Usage:**

   - Increased demand leads to higher CPU usage.

3. **CloudWatch Monitoring:**

   - CloudWatch monitors the CPU usage.
   - A CloudWatch alarm is triggered when the CPU usage exceeds the specified threshold.

4. **Triggering Scaling Activity:**

   - The alarm triggers a scaling activity in AWS Application Auto Scaling.
   - The desired capacity of the ECS service increases.

5. **Launching New Tasks:**

   - New tasks are created to handle the increased load.

6. **Scaling EC2 Instances (if using EC2 launch type):**

   - **ECS Capacity Provider:** Automatically scales the ASG by adding new EC2 instances when needed.
   - Ensures that there is sufficient capacity to run the new tasks.

By leveraging ECS Service Auto Scaling, you can ensure that your application scales efficiently in response to changes in demand, providing consistent performance while optimizing resource usage.

Updating an ECS Service with Rolling Updates

When updating an ECS service from one version to another (e.g., from v1 to v2), you can manage the process using rolling updates. This involves controlling the number of tasks started and stopped at a time, ensuring minimal disruption. Two key settings for rolling updates are the **Minimum Healthy Percent** and the **Maximum Percent**.

Key Settings:

- **Minimum Healthy Percent**: Defines the minimum number of tasks that must be running and healthy during the update. By default, it is set to 100%, meaning all tasks must remain healthy.
- **Maximum Percent**: Defines the maximum number of tasks that can run during the update. By default, it is set to 200%, allowing for the current number of tasks plus an equal number of new tasks.

Example Scenarios:

1. **Scenario 1: Minimum 50%, Maximum 100%**
    - **Initial Setup**: 4 tasks running (100% capacity)
    - **Steps**:
        1. **Terminate 2 tasks**: Running at 50% capacity.
        2. **Create 2 new tasks**: Back to 100% capacity with new tasks.
        3. **Terminate 2 old tasks**: Back to 50% capacity.
        4. **Create 2 new tasks**: Back to 100% capacity with all new tasks.
    - **Result**: Gradual replacement of old tasks with new tasks, maintaining at least 50% capacity at all times.
2. **Scenario 2: Minimum 100%, Maximum 150%**
    - **Initial Setup**: 4 tasks running (100% capacity)
    - **Steps**:
        1. **Create 2 new tasks**: Capacity increases to 150%.
        2. **Terminate 2 old tasks**: Back to 100% capacity.
        3. **Create 2 new tasks**: Capacity increases to 150% again.
        4. **Terminate 2 old tasks**: Back to 100% capacity with all new tasks.
    - **Result**: Maintains full capacity throughout the update, ensuring no downtime.

Workflow:

1. **Initiate Update**: Choose the new task definition version (e.g., v2).
2. **Rolling Update Process**:
    - **Minimum Healthy Percent** dictates the minimum capacity to maintain.
    - **Maximum Percent** allows the ECS to temporarily exceed normal capacity.
    - ECS starts and stops tasks in phases, ensuring minimum disruption.
3. **Completion**: All old tasks are replaced by new ones, adhering to the specified update strategy.

Using these settings, ECS ensures a smooth transition from the old version to the new version, maintaining service availability and performance. This methodical approach prevents significant disruptions during updates, providing a robust and reliable way to manage ECS services.

Solution Architectures with Amazon ECS

Here are a few solution architectures involving Amazon ECS and how they can be utilized with various AWS services:

1. ECS Tasks Invoked by EventBridge

- **Architecture**:
    - **ECS Cluster**: Backed by Fargate.
    - **S3 Buckets**: Users upload objects to these buckets.
    - **EventBridge**: Integrated with S3 to capture events.
    - **ECS Tasks**: Launched based on EventBridge rules.
- **Workflow**:
    1. **User Action**: Users upload objects to S3 buckets.
    2. **Event Trigger**: S3 triggers events that are sent to EventBridge.
    3. **EventBridge Rule**: Defines a rule to run ECS tasks when these events occur.
    4. **Task Execution**: ECS tasks are launched with associated IAM roles.
    5. **Processing**: ECS tasks retrieve and process objects from S3.
    6. **Storage**: Processed results are stored in DynamoDB.
- **Benefits**:
    - **Serverless Processing**: Fully serverless and scalable architecture.
    - **Seamless Integration**: Uses AWS services like S3, EventBridge, ECS (Fargate), and DynamoDB.
    - **Security**: Uses IAM roles to manage access securely.

2. EventBridge Schedule Trigger

- **Architecture**:
    - **ECS Cluster**: Backed by Fargate.
    - **EventBridge**: Schedule rules to trigger events.
- **Workflow**:
    1. **EventBridge Rule**: Set to trigger every hour (or any specified interval).
    2. **Task Launch**: EventBridge launches ECS tasks in the Fargate cluster.
    3. **Batch Processing**: Tasks execute batch processing (e.g., operations on files in S3).
    4. **Task Role**: ECS tasks have roles allowing them to access necessary AWS resources like S3.

- **Benefits**:
    - **Automated Scheduling**: Automated task execution at regular intervals.
    - **Serverless**: No need to manage underlying infrastructure.
    - **Scalable**: Fargate scales the underlying resources as needed.

3. ECS with SQS Queue

- **Architecture**:
    - **ECS Service**: Contains multiple ECS tasks.
    - **SQS Queue**: Receives and holds messages.
    - **ECS Service Auto Scaling**: Adjusts the number of tasks based on queue size.
- **Workflow**:
    1. **Message Arrival**: Messages are sent to an SQS queue.
    2. **Task Polling**: ECS tasks poll the SQS queue for messages.
    3. **Processing**: Tasks process messages from the queue.
    4. **Auto Scaling**: Based on the number of messages, ECS service auto-scales (adds/removes tasks).
- **Benefits**:
    - **Scalable Processing**: Dynamically adjusts processing power based on workload.
    - **Decoupled Architecture**: Components interact via SQS, reducing dependencies.
    - **Efficient Resource Utilization**: Automatically scales to handle varying loads.

4. EventBridge for ECS Cluster Events

- **Architecture**:
    - **EventBridge**: Monitors ECS cluster events.
    - **SNS Topic**: Sends notifications based on events.
- **Workflow**:
    1. **Event Monitoring**: EventBridge monitors ECS cluster events (e.g., task state changes).
    2. **Event Trigger**: Specific events (e.g., task stopped) trigger EventBridge rules.
    3. **Notification**: EventBridge rules send notifications to an SNS topic.
    4. **Alerting**: SNS topic sends emails to administrators or other endpoints.
- **Benefits**:
    - **Lifecycle Monitoring**: Provides visibility into container lifecycle events.
    - **Automated Alerts**: Automatically notifies administrators of significant events.
    - **Enhanced Management**: Helps in managing and maintaining ECS cluster health.

These solution architectures demonstrate how Amazon ECS can be effectively integrated with other AWS services to build scalable, reliable, and fully-managed applications. By leveraging Fargate, EventBridge,

SQS, and other AWS services, you can create robust solutions that automatically respond to events, schedule tasks, and manage workloads efficiently.

Amazon ECS Task Definitions: In-Depth Explanation

Amazon ECS task definitions are essential in telling the ECS service how to run one or more Docker containers. They are defined in JSON format, and the console provides a UI to help create this JSON. Key elements within a task definition include image name, port bindings, memory, CPU, environment variables, networking information, IAM role, and logging configuration.

Key Components of ECS Task Definitions

1. **Image Name**: Specifies the Docker image to use for the container.
2. **Port Binding**:
    - **Container Port**: The port on the container where the application is running.
    - **Host Port** (for EC2 launch type): The port on the host (EC2 instance) that maps to the container port.
3. **Memory and CPU**: The amount of memory and CPU resources required for the container.
4. **Environment Variables**: Configuration settings passed to the container.
5. **Networking Information**: Defines how the container interacts with other network resources.
6. **IAM Role**: Permissions that allow the container to interact with other AWS services.
7. **Logging Configuration**: Setup for logging, such as CloudWatch Logs.

Example Scenario

For instance, consider an ECS task running an Apache HTTP server on an EC2 instance. The task definition would include:

- **Container Port**: 80 (where the HTTP server is running).
- **Host Port**: Mapped to a port on the EC2 instance (e.g., 8080).

This setup enables external access to the HTTP server running in the container through the EC2 instance's port.

Multiple Containers per Task Definition

You can define up to 10 containers per task definition, useful for creating sidecar containers that assist with logging, monitoring, or other ancillary functions.

Deep Dive into Port Mapping

- **Dynamic Host Port Mapping (EC2 launch type)**:

    - If only the container port is defined and the host port is set to 0, the host port will be dynamically assigned.
    - The Application Load Balancer (ALB) automatically knows how to route traffic to the correct dynamically assigned host ports, a feature not available with the Classic Load Balancer.

- **Fargate Launch Type**:
    - Each ECS task receives a unique private IP address.
    - No host port definition is needed; only the container ports are defined.
    - ALB connects to these tasks using their private IP addresses and defined container ports.

IAM Roles in ECS

IAM roles are assigned at the task definition level, not the service level. Each task within a service inherits the IAM role defined in the task definition, allowing tasks to securely access AWS resources like S3 or DynamoDB.

Environment Variables

- **Hard-Coded Variables**: Directly specified within the task definition.
- **Sensitive Variables**: Use SSM Parameter Store or Secrets Manager to store sensitive information such as API keys or passwords. These values are fetched at runtime and injected into the container as environment variables.
- **Bulk Loading**: Environment variables can also be loaded from an S3 bucket file.

Sharing Data Between ECS Tasks

To share data between multiple containers within a task, you can use data volumes:

- **Bind Mounts**: Shared storage volumes defined in the task definition.
    - **EC2 Tasks**: Bind mounts use the EC2 instance storage.
    - **Fargate Tasks**: Bind mounts use ephemeral storage, tied to the container's lifecycle.

For example, application containers can write logs to a shared storage location, and a sidecar container can read these logs to forward them to a centralized logging service.

Summary

Amazon ECS task definitions provide a comprehensive way to define how Docker containers run in an ECS cluster. Key components include container configurations, resource requirements, environment variables, IAM roles, and networking setups. Understanding these elements and their interactions is crucial for effectively managing ECS-based applications.

Creating an Amazon ECS Task Definition: Detailed Explanation

When creating an Amazon ECS task definition, several options and configurations are available. Here's a comprehensive guide to setting up a task definition.

1. Family Name

- Example: `wordpress`
- This identifies the task definition family.

2. Infrastructure Requirements

- **Fargate**: Requires specifying compatible CPU and memory values.
- **EC2 Instances**: Allows flexible values for memory and CPU.

3. Network Mode

- **Fargate**: Must use `AWS VPC`.
- **EC2**: Can choose advanced network modes.

4. IAM Roles

- **Task Role**: Allows containers to make API calls to AWS services. Crucial for accessing resources like S3 or DynamoDB.
- **Task Execution Role**: Allows the ECS container agent to make AWS API requests on your behalf. Essential for pulling container images and storing logs.

5. Containers Configuration

- You can define multiple containers in a task definition.
- **Essential Container**: If marked as essential, the task stops if this container fails.
- **Container Name and Image**: For example, `wordpress` with an image URI like `wordpress`.
- **Private Registry Authentication**: Use AWS Secrets Manager for authentication if pulling from a private registry.

6. Port Mapping

- Define container ports and corresponding host ports.
- Specify protocol (HTTP, HTTP2, GRPC, or none).

7. Resource Limits

- Set VCPUs and memory limits, both hard and soft limits.

8. Environment Variables

- **Static Variables**: Directly hardcoded in the task definition.
- **Sensitive Variables**: Use AWS Secrets Manager or SSM Parameter Store for values like API keys or passwords.
- **Bulk Loading**: Load variables from a file stored in Amazon S3.

9. Logging Configuration

- Options include CloudWatch, Splunk, Firehose, Kinesis Stream, OpenSearch, or S3.
- Use AWS FireLens for flexible log routing.
- Define log group, region, and stream prefix if using CloudWatch.

10. Health Checks

- Define health check parameters to ensure containers are running correctly.

11. Timeouts

- **Start Timeout**: Maximum time to allow a container to start.
- **Stop Timeout**: Time allowed for a container to stop gracefully.

12. Docker Configuration

- Add Docker labels and other specific Docker settings.

13. Storage Configuration

- **Bind Mounts**: Use host storage for container data.
- **EFS (Elastic File System)**: Mount an EFS file system to containers.
- **Mount Points**: Define where volumes are mounted in the container.

14. Monitoring and Metrics

- Enable AWS X-Ray tracing through AWS Distro for Open Telemetry sidecar.
- Collect and send metrics to CloudWatch or Managed Service for Prometheus.

15. Review and Create

- Review all configurations in JSON format.
- Create the task definition and potentially modify and revise as needed.

By following these steps, you can create a detailed and fully functional ECS task definition using the AWS console. This setup allows you to manage and deploy containerized applications effectively on AWS.

ECS Task Placement Strategies and Constraints: Detailed Explanation

Amazon ECS (Elastic Container Service) provides various strategies and constraints to control how tasks are placed on EC2 instances within a cluster. Understanding these concepts is crucial for optimizing resource utilization, ensuring high availability, and managing costs efficiently.

1. Task Placement Process

When ECS needs to place or scale tasks within an EC2 cluster, it follows a structured process:

- **Identify Instances**: ECS first identifies EC2 instances that meet the CPU, memory, and port requirements specified in the task definition.
- **Apply Constraints**: It then considers any task placement constraints defined for the task.
- **Apply Strategies**: Finally, ECS applies a task placement strategy to select the optimal instance based on the strategy's criteria (e.g., binpack, spread, or random).

2. Task Placement Strategies

Task placement strategies dictate how ECS chooses an instance for placing new tasks. Here are the main strategies:

- **Binpack**: Maximizes instance utilization by placing tasks based on the least available amount of CPU or memory. This minimizes the number of instances in use and can result in cost savings.

  JSON Definition:

  ```
  {
    "type": "binpack",
    "field": "memory"
  }
  ```

- **Spread**: Spreads tasks evenly across instances, typically in different availability zones (AZs) or instance IDs to improve fault tolerance and high availability.

  JSON Definition:

  ```
  {
    "type": "spread",
    "field": "availability-zone"
  }
  ```

- **Random**: Places tasks randomly across available instances without any specific logic or optimization.

  JSON Definition:

  ```
  {
    "type": "random"
  }
  ```

3. Task Placement Constraints

Constraints provide rules that restrict where tasks can be placed. These are particularly useful for compliance, performance optimization, or regulatory requirements.

- **DistinctInstance**: Ensures that each task is placed on a different container instance. This is useful for redundancy and minimizing risk.

  JSON Definition:

  ```
  {
    "type": "distinctInstance"
  }
  ```

- **MemberOf**: Specifies that tasks should be placed only on instances that satisfy an expression in the Cluster Query Language (CQL). This allows precise control over task placement based on instance attributes such as instance type or custom tags.

  Example JSON:

  ```
  {
    "type": "memberOf",
    "expression": "attribute:instance-type =~ t2.*"
  }
  ```

4. Mixing Strategies and Constraints

ECS allows mixing different strategies and constraints to tailor placement behavior according to specific needs. For example, combining spread across availability zones with a binpack strategy based on memory can optimize both fault tolerance and resource utilization.

5. Use Cases and Considerations

- **Cost Optimization**: Binpack strategy reduces the number of EC2 instances in use, thus lowering costs by maximizing utilization.

- **High Availability**: Spread strategy ensures tasks are distributed across different AZs or instances, reducing the impact of failures.
- **Compliance**: Constraints like distinctInstance can ensure regulatory requirements are met by preventing co-location of sensitive tasks.

Conclusion

Understanding ECS task placement strategies and constraints is essential for designing efficient, resilient, and cost-effective containerized applications on AWS ECS. These concepts not only optimize resource utilization but also enhance the reliability and scalability of ECS deployments.

Steps to Destroy an ECS Service and Cluster

When you're done using an ECS service and want to clean up your environment, follow these steps to ensure everything is properly deleted.

1. Stop the Service

- **Set Desired Tasks to Zero**:
    - Navigate to your service in the ECS console.
    - Update the service and set the desired task count to zero. This ensures there are no running tasks.
- **Delete the Service**:
    - Select the service and click on "Delete Service."
    - Confirm by typing "Delete" when prompted.

2. CloudFormation Stack Deletion

- **Wait for Deletion**:
    - Deleting the service triggers CloudFormation to start deleting the stack it created.
    - CloudFormation will remove the ECS service, LoadBalancer Listener, LoadBalancer, security groups, and target groups.
    - This process can take some time, so wait until it's fully completed.

3. Delete the ECS Cluster

- **Delete the Cluster**:
    - Once the service is deleted, navigate to the ECS cluster.
    - Click on "Delete Cluster" and confirm.
    - CloudFormation will handle the deletion of the cluster infrastructure, including the capacity provider, auto-scaling group, and launch templates.

4. Optional: Deregister Task Definitions

- **Deregister Task Definitions**:
    - Task definitions do not incur costs and can be left as they are.
    - If you prefer to remove them, select the task definition.
    - Choose "Actions" and then "Deregister" to remove the task definition.

By following these steps, you will ensure that all components related to your ECS service and cluster are properly deleted, preventing any unnecessary charges or resource usage.

Introduction to Amazon ECR (Elastic Container Registry)

Amazon ECR, or Elastic Container Registry, is a service provided by AWS to store, manage, and deploy Docker images. Here's a concise overview of its key features and functionalities:

Storage and Management of Docker Images

- **Private and Public Repositories**:
  - **Private Repositories**: Store Docker images accessible only to your AWS account or specific accounts.
  - **Public Repositories**: Publish images to the Amazon ECR public gallery, making them available publicly.

Integration with Amazon ECS

- **Seamless Integration**:
  - Amazon ECR integrates seamlessly with Amazon ECS (Elastic Container Service), enabling easy deployment of Docker images.
  - **Storage**: Behind the scenes, Docker images are stored in Amazon S3, ensuring durability and scalability.

Access and Security

- **IAM Role-Based Access**:
  - To pull Docker images from ECR, ECS instances (like EC2) are assigned IAM roles. These roles grant the necessary permissions to access ECR.
  - **IAM Policies**: All access to ECR is controlled through IAM policies, ensuring secure and controlled access to Docker images.

Advanced Features

- **Image Vulnerability Scanning**:
  - ECR supports scanning of Docker images for vulnerabilities, helping maintain security and compliance.
- **Versioning and Tags**:
  - Manage different versions of your Docker images using tags, ensuring you can track and use specific image versions.
- **Image Lifecycle Management**:
  - Define rules to manage the lifecycle of images in your repositories, such as automatically deleting old images.

Use Case

- **Pulling Images**:

- An EC2 instance in your ECS cluster can pull Docker images stored in ECR. The instance uses its assigned IAM role to authenticate and retrieve the images.
- Once pulled, the Docker images can be used to start containers on the EC2 instance.

Amazon ECR provides a robust solution for storing and managing Docker images, integrating smoothly with ECS, and offering advanced features for image management and security. This makes it an essential tool for containerized application deployments on AWS.

Using the CLI to Pull and Push Images to Amazon ECR

Amazon ECR (Elastic Container Registry) allows you to store and manage Docker images on AWS. Here's a step-by-step guide on how to use the CLI to pull and push Docker images to Amazon ECR.

1. Login to Amazon ECR

To start using ECR with your Docker CLI, you need to log in. This involves getting Docker credentials for your Docker CLI.

- **Get Login Password**:

```
aws ecr get-login-password --region <your-region>
```

- **Login to Docker**:

```
aws ecr get-login-password --region <your-region> | docker login --
username AWS --password-stdin <aws_account_id>.dkr.ecr.
<region>.amazonaws.com
```

2. Create a Private Repository on ECR

- Navigate to the Amazon ECR service in the AWS Management Console.
- Create a new repository (e.g., `demostephane`).
- You have options for tag immutability, image scanning, and encryption, but these can be left disabled for a basic setup.

3. Push an Image to ECR

Once the repository is created, you can push images to it.

- **Build and Tag the Image**:
  Assuming you have an image named `nginxdemos/hello`:

```
docker pull nginxdemos/hello
```

- **Tag the Image**:

```
docker tag nginxdemos/hello:latest <aws_account_id>.dkr.ecr.
<region>.amazonaws.com/demostephane:latest
```

- **Push the Image**:

```
docker push <aws_account_id>.dkr.ecr.
<region>.amazonaws.com/demostephane:latest
```

4. Pull an Image from ECR

If you want to pull an image from your ECR repository:

- **Pull the Image**:

```
docker pull <aws_account_id>.dkr.ecr.
<region>.amazonaws.com/demostephane:latest
```

5. Manage IAM Permissions

Ensure that the IAM role assigned to your EC2 instance or CLI user has the necessary permissions to access ECR. If you encounter permission errors, check your IAM policies.

Example Workflow

1. **Login to ECR**:

```
aws ecr get-login-password --region us-west-2 | docker login --username
AWS --password-stdin 123456789012.dkr.ecr.us-west-2.amazonaws.com
```

2. **Create and Tag the Image**:

```
docker pull nginxdemos/hello
docker tag nginxdemos/hello:latest 123456789012.dkr.ecr.us-west-
2.amazonaws.com/demostephane:latest
```

3. **Push the Image**:

```
docker push 123456789012.dkr.ecr.us-west-
2.amazonaws.com/demostephane:latest
```

4. **Pull the Image**:

```
docker pull 123456789012.dkr.ecr.us-west-
2.amazonaws.com/demostephane:latest
```

Additional Features of Amazon ECR

- **Image Vulnerability Scanning**: Supports scanning images for vulnerabilities.
- **Versioning and Tags**: Manage different versions of images using tags.
- **Image Lifecycle Management**: Define rules to manage the lifecycle of images in your repositories.

Using these commands and steps, you can effectively manage Docker images within Amazon ECR, ensuring secure and efficient container deployments on AWS.

AWS Copilot: Simplifying Containerized Application Management

AWS Copilot is a command line interface (CLI) tool designed to simplify the process of building, releasing, and operating production-ready containerized applications on AWS. It focuses on abstracting the complexities involved in managing infrastructure, allowing developers to concentrate on building their applications.

Key Features of AWS Copilot

1. **Simplified Deployment**:
    - **Abstracted Infrastructure**: AWS Copilot handles the underlying infrastructure setup, including ECS, VPC, ELB, and ECR, reducing the need for manual configuration.
    - **Deployment Environments**: Easily deploy applications to AppRunner, ECS, and Fargate using straightforward CLI commands.
2. **Automated Deployment Pipelines**:
    - **CodePipeline Integration**: Copilot can be integrated with AWS CodePipeline to automate container deployments with a single command, streamlining the deployment process.
3. **Multi-Environment Deployment**:
    - **Multiple Environments**: Supports deploying applications across multiple environments, such as development, testing, and production, ensuring consistency and reliability.
4. **Operational Insights**:
    - **Monitoring and Troubleshooting**: Provides tools for troubleshooting, accessing logs, and checking the health status of applications, facilitating efficient operations and maintenance.
5. **Microservice Architecture Support**:
    - **Application Architecture**: Use CLI commands or YAML files to describe the architecture of applications in a microservice-oriented manner, ensuring modular and scalable designs.
6. **Containerization and Deployment**:
    - **Containerize Applications**: Simplifies the process of containerizing applications and deploying them to the specified environments, ensuring they are production-ready.
    - **Well-Architected Setup**: Deployments are right-sized and configured to scale automatically, adhering to best practices.

Workflow with AWS Copilot

1. **Describe Your Application**:

- Use the CLI or a YAML file to define the architecture of your application, including services and their configurations.

2. **Deploy the Application**:
    - Utilize AWS Copilot CLI commands to containerize your application and deploy it to ECS, Fargate, or AppRunner.
3. **Manage and Monitor**:
    - Gain insights into your application's performance and health using built-in tools for logging and troubleshooting.
4. **Automate Deployments**:
    - Integrate with AWS CodePipeline to automate the deployment process, ensuring continuous delivery and integration.

Summary

AWS Copilot streamlines the deployment and management of containerized applications by abstracting the complexities of AWS infrastructure. It provides a simple, CLI-driven approach to deploy applications to ECS, Fargate, and AppRunner, supports multi-environment deployments, and offers tools for effective monitoring and troubleshooting. With AWS Copilot, you can focus on building your applications while it handles the infrastructure setup and operational aspects.

AWS Copilot: Practical Guide to Deploying an Application

AWS Copilot is a CLI tool that simplifies deploying containerized applications on AWS. This guide demonstrates how to use Copilot to deploy an application using AWS Cloud9.

Steps to Deploy an Application Using AWS Copilot

1. Set Up Cloud9 Environment

1. **Create a Cloud9 Environment**:
   - Go to Cloud9 in the AWS Management Console.
   - Create a new environment named `DemoCloud9`.
   - Use a `t2.micro` instance with Amazon Linux 2.
   - Set a timeout of 30 minutes.
   - Access using Systems Manager.
2. **Open Cloud9**:
   - Click on the created environment to open it.

2. Install Copilot and Docker

1. **Install AWS Copilot CLI**:
   - Check if Copilot is installed by typing `copilot` in the terminal.
   - If not installed, run the installation command:

     ```
     curl -Lo copilot https://github.com/aws/copilot-cli/releases/latest/download/copilot-linux && chmod +x copilot && sudo mv copilot /usr/local/bin/copilot
     ```

   - Verify the installation with:

     ```
     copilot --version
     ```

2. **Verify Docker Installation**:
   - Ensure Docker is installed by typing `docker` in the terminal.

3. Clone Sample Repository

1. **Clone Repository**:
   - Clone a repository containing a sample Docker application:

```
git clone https://github.com/aws-samples/aws-copilot-sample-
service.git example
```

- Navigate to the example directory:

```
cd example/
```

4. Initialize and Deploy with Copilot

1. **Initialize Copilot**:
   - Run the initialization command:

```
copilot init
```

   - Follow the prompts:
     - Application name: `copilot-guide`
     - Workload type: `Load Balanced Web Service`
     - Service name: `web-app`
     - Dockerfile location: `./Dockerfile`
2. **Create Environment**:
   - Initialize the environment:

```
copilot env init --name prod --profile default
```

   - Select default configuration for new VPC, subnets, ECS cluster, and IAM roles.
3. **Deploy Environment**:
   - Deploy the environment:

```
copilot env deploy --name prod
```

4. **Deploy Service**:
   - Deploy the service:

```
copilot deploy
```

5. **Verify Deployment**:
    - Go to ECS in the AWS Management Console and verify that the `copilot-guide-prod` service is running.
    - Access the application via the load balancer's DNS name.

5. Clean Up Resources

1. **Delete Application**:
    - Delete the Copilot application:

```
copilot app delete
```

2. **Remove User and Access Keys**:
    - Delete the user created for Copilot in IAM.
3. **Delete Cloud9 Environment**:
    - Go to Cloud9 in the AWS Management Console and delete the environment.

Summary

AWS Copilot greatly simplifies deploying containerized applications by automating the setup of underlying infrastructure and providing easy-to-use commands for deployment and management. This guide walks through setting up a Cloud9 environment, installing necessary tools, initializing and deploying an application, and cleaning up resources afterward. By following these steps, you can efficiently deploy applications with minimal manual configuration.

Amazon Elastic Kubernetes Service (EKS) Overview

Amazon Elastic Kubernetes Service (EKS) is a managed service that allows you to run and manage Kubernetes clusters on AWS. Here's a detailed explanation of EKS and its features.

What is Kubernetes?

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications. It is widely used across different cloud providers, offering a standardized way to manage containers.

EKS Launch Modes

1. **EC2 Launch Mode**:
    - Deploy worker nodes as EC2 instances.
2. **Fargate Mode**:
    - Deploy serverless containers in an EKS cluster.

Use Cases for EKS

- Organizations already using Kubernetes on-premises or on other cloud platforms.
- Preference for Kubernetes API to manage containerized applications.
- Simplifies cloud migration by providing a consistent Kubernetes experience across different cloud environments.

EKS Architecture

- **VPC Setup**: Typically spans three availability zones (AZs) with a mix of public and private subnets.
- **EKS Worker Nodes**: Deployed as EC2 instances running EKS Pods (similar to ECS tasks).
- **Auto Scaling**: Managed by Auto Scaling groups to dynamically adjust the number of nodes based on demand.
- **Load Balancers**: Private or public load balancers can be set up to expose EKS services.

Node Types in EKS

1. **Managed Node Groups**:
    - AWS manages the EC2 instances for you.
    - Nodes are part of an Auto Scaling group.
    - Supports On-Demand and Spot Instances.
2. **Self-Managed Nodes**:
    - Provides more customization and control.
    - You create and register the nodes to an EKS cluster.

    - Nodes can be part of an ASG.
    - Can use Amazon EKS Optimized AMI or build your own AMI.
    - Supports On-Demand and Spot Instances.

3. **Fargate Mode**:

    - No node management required.
    - Run containers without provisioning or managing EC2 instances.

Storage in EKS

You can attach data volumes to your EKS cluster using the Container Storage Interface (CSI) compliant driver by specifying a StorageClass manifest.

- **Amazon EBS**
- **Amazon EFS** (works with Fargate)
- **Amazon FSx for Lustre**
- **Amazon FSx for NetApp ONTAP**

Summary

Amazon EKS provides a managed Kubernetes environment, supporting different deployment modes and storage options, making it a versatile solution for running containerized applications on AWS.