ECS

- Install AWS CLI

https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

- ECR - Amazon Elastic Container Registry - Docker container registry like DockerHub

In ECS, a task definition specifies how Docker containers should be configured and run. Tasks are instances of these definitions, representing actual running containers. ECS services then manage these tasks, ensuring the specified number of instances are running and handling scaling and availability. So, task definitions define the container setup, tasks are the running instances based on those definitions, and services manage the lifecycle and scaling of those tasks.

1. ECS Service:
   - Manages groups of tasks, ensuring a specified number run.
   - Handles scaling and availability.
2. Task Definition:
   - Configuration for running a Docker container.
   - Defines Docker image, resources, networking, etc.
3. Task:
   - An instance of a task definition running in ECS.
   - Represents a containerized application or workload.

Basic Demo Code

```
//index.js

const express = require('express');
const app = express();
const port = 3000;

// Define a route
app.get('/', (req, res) => {
res.send('Congratulations HHLD Folks - we are in ECS Demo!');
});

app.get('/health', (req, res) => {
   res.status(200).send('ECS Demo is healthy');
});


// Start the server
app.listen(port, () => {
console.log(`Server is listening at http://localhost:${port}`);
});
```

```
# Dockerfile
FROM --platform=linux/amd64 node:latest
WORKDIR /app
COPY . /app
RUN npm install
EXPOSE 3000
CMD node index.js
```

```
.gitignore
node_modules
```

https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-container-image.html
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/getting-started-fargate.html

- Push your image to Amazon Elastic Container Registry
    - Create an AWS ECR repository to store your image.
    - Tag the image
    - Run the aws ecr get-login-password command
    - Push the image to Amazon ECR
- Create the cluster
- Create a task definition
    - Specify image URL from ECR repository
    - Add port mapping
- Go back to the cluster and Create a service using the task definition
    - Mention task and no of tasks
    - Add Application Load Balancer
    - Add auto scaling and policy for that
    - Notice the no of tasks in your created service
    - Go to the LB you created
    - Check security group of LB and allow inbound rule for HTTP
    - Checkout the public URL of LB
- Rolling update-
    - Make change in your code and push it to ECR
    - Go back to service and update it - force new deployment
    - Check the tasks and their health statuses (keep refreshing)
    - Check the same public URL of LB

CICD using GitHub

- Create Git Repo and push code
- Go to Actions > Deploy to Amazon ECS
- Change env values from AWS
- For task definition => Download JSON from AWS and upload file on GitHub
- In GitHub, go to settings > secrets and variables > actions
- Add repo secrets from AWS => Profile > Security credentials > Access Keys
- Make change in your GitHub code (or push changes)
- Go to Actions and see Deployment

Continuous Integration (CI):

1. **Frequent Code Integration:** Developers frequently merge their code changes into a shared repository, usually multiple times a day.

2. **Automated Builds:** Automated processes are triggered whenever code changes are submitted, such as compiling code, running tests, and generating artifacts.

3. **Early Detection of Issues:** Automated tests help identify bugs and integration issues early in the development cycle, preventing them from snowballing into larger problems.

4. **Integration Feedback:** Developers receive immediate feedback on the quality of their code changes, allowing them to address issues quickly.

Continuous Delivery (CD):

1. **Automated Deployment Pipeline:** A series of automated steps that take code from the repository to production, including testing, staging, and deployment.

2. **Consistent Deployment:** Ensures that every code change that passes through the pipeline can be deployed to production environments consistently and reliably.

3. **Fast Feedback Loop:** Enables rapid feedback on the deployability of code changes, reducing the time between writing code and delivering it to users.

4. **Incremental Updates:** Allows for incremental updates to be deployed to production frequently, rather than large, infrequent releases, reducing risk and enabling faster innovation.

https://docs.aws.amazon.com/whitepapers/latest/cicd_for_5g_networks_on_aws/cicd-on-aws.html

CodeCommit - GitHub/ BitBucket
CodeBuild
CodeDeploy
CodePipeline

CSR, SSR, SSG, and ISR are all techniques for rendering web pages, but they differ in where the rendering happens and how they handle data updates.

- **Client-side Rendering (CSR):**

This approach involves the server sending a basic HTML shell to the browser. The browser then fetches data and uses JavaScript to dynamically generate the full page content.

  - The server sends a basic HTML shell to the browser.
  - The browser then fetches the necessary data and uses JavaScript to dynamically generate the full page content.
  - **Pros:** Fast initial load, feels very responsive as interactions happen on the client-side.
  - **Cons:** Not ideal for SEO (Search Engine Optimization) as search engines can't easily see the content, can be slower for pages with a lot of data.

- **Server-side Rendering (SSR):**
  - The server generates the full HTML content for each request, including any data needed.
  - The fully rendered HTML is then sent to the browser.
  - **Pros:** Excellent for SEO, good for performance for complex interactions.
  - **Cons:** Slower initial load time compared to CSR as the server needs to render the page on each request.

- **Static Site Generation (SSG):**
  - The content is pre-rendered at build time, creating static HTML files.
  - These static files are then served directly by the server.
  - **Pros:** Very fast initial load times, excellent for SEO.
  - **Cons:** Content updates require rebuilding the entire site, not ideal for frequently changing content.

- **Incremental Static Regeneration (ISR):**
  - Similar to SSG, content is pre-rendered at build time.
  - However, ISR allows for updates at regular intervals or on-demand.
  - **Pros:** Combines the benefits of SSG (fast load times, SEO) with the ability to keep content fresher than pure SSG.
  - **Cons:** Content may not be the absolute latest if updated just before a user sees a cached version.

Choosing the best approach depends on your specific needs. Here's a general guideline:

- Use CSR for highly dynamic applications where SEO is not a major concern.
- Use SSR for applications that need good SEO and complex interactions.
- Use SSG for mostly static content where fast load times are crucial.
- Use ISR for content that is mostly static but needs to be occasionally refreshed.

https://nextjs.org/docs/pages/building-your-application/rendering
https://nextjs.org/docs/app/building-your-application/rendering/composition-patterns
https://nextjs.org/learn/react-foundations/server-and-client-components

Hydration:

Process of transforming server-rendered HTML into a fully interactive React application in the browser.

Steps:

1. **Server Renders HTML:**
   - Fetches data and renders initial HTML with placeholders for dynamic elements.

2. **Browser Receives HTML:**
   - Displays static content from server-rendered HTML.
   - Downloads and executes JavaScript bundle.

3. **Hydration (JavaScript execution):**
   - Reconstructs React component tree based on server markup.
   - Attaches event listeners and interactive features to DOM elements.
   - Fetches additional data (if needed) and updates DOM.

Outcome:

- Creates fully functional and interactive React application in the browser.

Benefits:

- Fast initial load due to pre-rendered HTML.
- Improved SEO as search engines can crawl initial content.

Smooth transition from static HTML to interactive app.

Hydration errors in Next.js arise when there's a mismatch between the initial HTML content rendered on the server and the HTML that the client-side JavaScript expects to build the application.

```
"use client" // comment and uncomment to see if log comes on webbrowser or
not
const x = Math.random();

export default function Home() {
 console.log('logging random num' , x);
 return (
   <div>
     Hello
   </div>
 );
}
```

```
"use client"
const x = Math.random();
export default function Home() {
 console.log('logging random num' , x);
 return (
   <div suppressHydrationWarning> // This is just one way
     {x}
   </div>
 );}
```

How rendering affects SEO

How SEO Works:

1. **Crawling**: Search engine bots, also known as crawlers or spiders, visit web pages by following links from one page to another. They discover new and updated content to be indexed through this process.
2. **Indexing**: After crawling, search engines analyze the content of web pages and store it in their index. Indexing involves parsing and understanding the content, including text, images, and metadata, to determine relevance and ranking for specific search queries.
3. **Ranking**: When a user enters a search query, search engines retrieve relevant pages from their index and rank them based on various factors such as relevance, authority, and user experience. Pages with higher rankings are displayed prominently in search results.
4. **Displaying Results**: Finally, search engines display the most relevant and authoritative pages in response to the user's query, along with titles, snippets, and URLs to help users find the information they need.

SEO is not as efficient during client-side rendering (CSR) compared to server-side rendering (SSR) or static site generation (SSG) for several reasons:

1. **Initial HTML Content**: With SSR and SSG, the server generates the full HTML content of the page, including its metadata, structure, and textual content. This complete HTML is readily available to search engine crawlers when they index the website, allowing them to understand the page's content and relevance to specific search queries.
2. **JavaScript Execution**: In CSR, the initial HTML sent to the client's browser is often minimal, containing placeholders or loading indicators for content that will be dynamically generated by JavaScript. Search engine crawlers may not execute JavaScript or may have limited capabilities to do so, resulting in incomplete or inaccurate indexing of the page's content.
3. **Time to Index**: Pages rendered client-side may take longer to index compared to SSR or SSG pages because search engine crawlers need to wait for JavaScript to execute and dynamically load content before indexing it. This delay can impact the freshness and relevance of content in search results.
4. **Metadata Handling**: Metadata such as title tags, meta descriptions, and canonical URLs are essential for SEO, as they provide valuable information to search engines about the page's content and purpose. While CSR frameworks often provide ways to update metadata dynamically, ensuring consistent and accurate metadata handling can be more complex compared to SSR or SSG, where metadata is generated server-side.