2024 – RAPIFY CLOUD - EMPER BACH

CI/CD Pipeline
For ReactJs Project on GitLab
Using AWS Services (ECR, Fargate, EC2)

Table of Contents

I. Introduction

This article is your guide to building a streamlined CI/CD pipeline for deploying your ReactJS website. Whether you're a developer seeking automation or a cloud engineer architecting a robust CI/CD solution, the powerful combination of GitLab and AWS services can help you achieve efficiency and reliability.

By leveraging GitLab's user-friendly interface and AWS's comprehensive suite of cloud services, you can automate the entire development and deployment process, from committing code changes to delivering your website to production. This approach offers significant benefits, including:

- **Reduced deployment time and effort**: Manual deployments are time-consuming and error prone. CI/CD automates the process, freeing you to focus on other tasks.
- **Improved software quality**: Automated testing ensures your code is functional and secure before deployment.
- **Faster response to changes**: CI/CD allows you to quickly respond to user feedback and security vulnerabilities.
- **Increased developer productivity**: By eliminating manual tasks, developers can focus on building new features and improvements.

Get ready to explore the exciting world of CI/CD for your ReactJS website!

II. Clarify Project: CI/CD Pipeline

1. CI/CD in Software Development

Imagine a popular social media app experiencing a sudden surge in users. Without CI/CD, scaling the app to accommodate the increased load would involve manual code edits, testing, and deployments – a slow and error-prone process. CI/CD automates this process, allowing the app to seamlessly scale and remain responsive to user demands.

2. Benefits of CI/CD

- **Reduced time and effort**: Automate building, testing, and deploying, freeing up valuable development time.
- **Improved quality**: Catch and fix bugs early in the development cycle through automated testing.
- **Faster updates**: Deliver new features and fixes to users quickly and efficiently.
- **Enhanced security**: Automate security scans to identify and address vulnerabilities early.
- **Greater collaboration**: Foster a streamlined workflow for development teams.

3. CI/CD Pipeline

A CI/CD pipeline is a series of automated stages that transform your code into a running website. Here's a typical breakdown

a. Continuous Integration (CI)

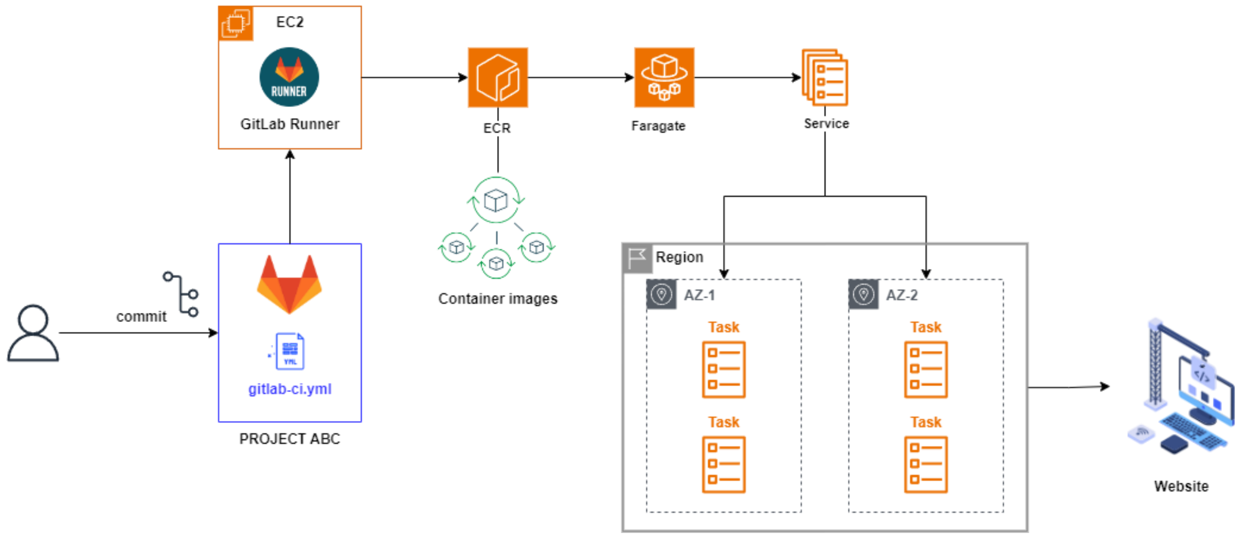
- Code changes are automatically pulled from your Git repository.
- Unit tests, code quality checks, and integration tests are run.
- If successful, the code is built into a deployable artifact.

b. Continuous Delivery (CD)

- The artifact is deployed to a staging environment for manual testing and approval.
- Upon approval, the artifact is deployed to the production environment.
- Performance and health checks are continuously run.

III. Architecture

1. Diagram



2. Walkthrough

a. Services

- **GitLab**: Provides version control for your code and facilitates the CI/CD pipeline through the gitlab-ci.yml file. It triggers the pipeline upon code changes and manages the overall workflow.
- **EC2**: Serves as a runner machine dedicated to executing your CI/CD jobs within the GitLab Runner framework. It runs tasks like building code, creating container images, and deploying applications.

- **ECR**: Acts as a secure repository to store container images built from your source code. ECR manages the access and versioning of these images.
- **Fargate**: Enables deployment of containerized applications without managing server infrastructure. Fargate provides a serverless environment for running your application containers.

b. Workflow

Step 1: Developer Code Changes

Developers modify the project's source code and push the changes to the GitLab repository.

Step 2: GitLab Trigger and Pipeline Creation

GitLab detects the code push and triggers a CI/CD pipeline based on the configurations defined in your `.gitlab-ci.yml` file.

Step 3: Container Image Build and Push

The GitLab Runner on the EC2 instance builds the container image from the source code using tools like Docker. Once built, the runner pushes the image to the ECR repository, making it accessible for deployment.

Step 4: Deployment Update

The GitLab Runner triggers a stage in the pipeline to pull the latest image version from ECR. The updated image is then used to deploy your application to Fargate, updating the running service with the latest changes. This is a simplified overview of the workflow. Your specific implementation may involve additional steps or tools depending on your specific requirements.

IV. Implement solution

1. Setup gitlab-ci.yml file

The `.gitlab-ci.yml` file plays a crucial role in orchestrating GitLab's CI/CD pipelines. This YAML configuration acts as a blueprint, meticulously outlining the automated tasks executed throughout your development and deployment process.

Within this article, we aim to establish a two-phase CI/CD pipeline:

Phase 1: Build

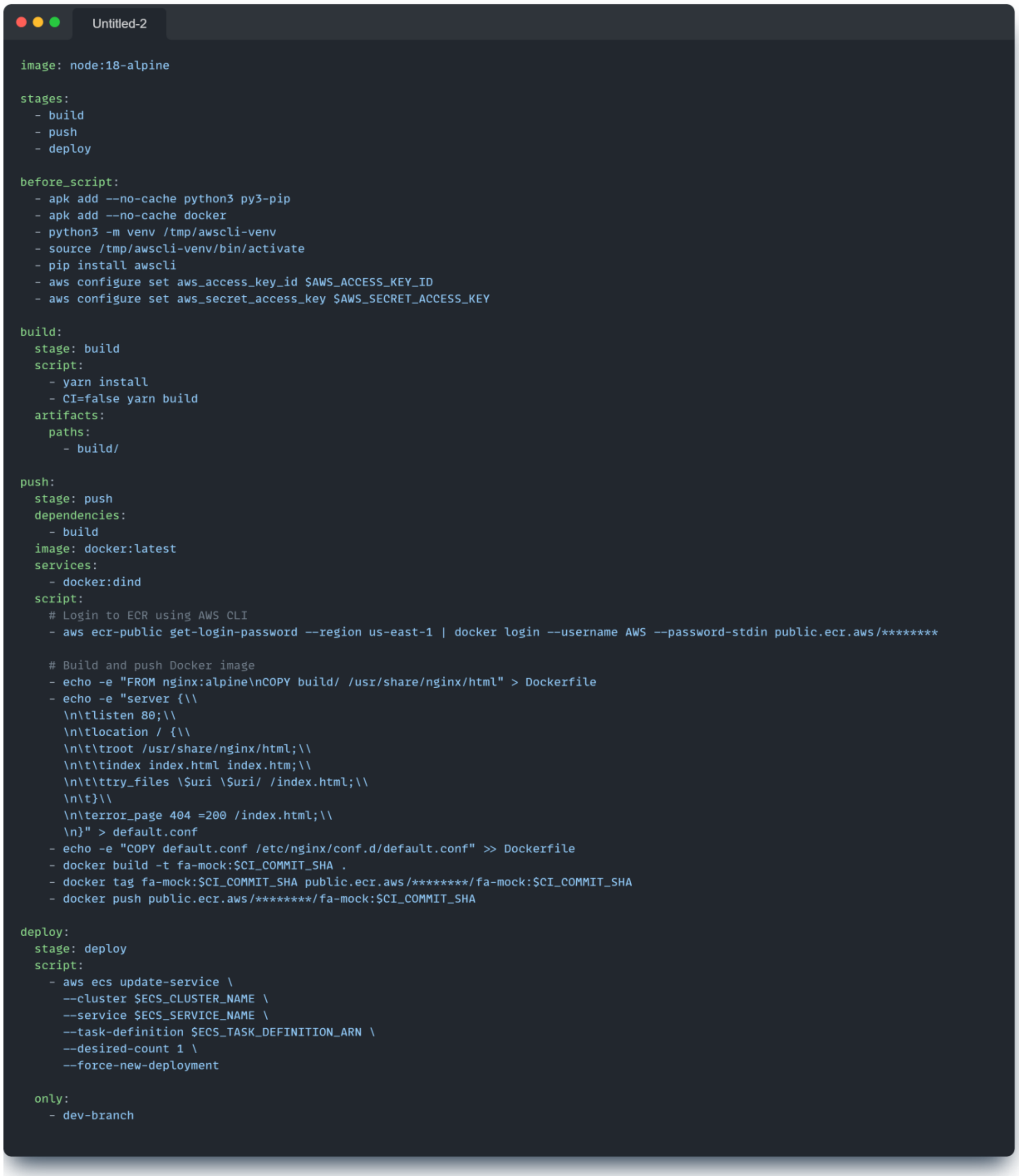
This phase meticulously constructs the application from its source code.

Phase 2: Push and Deploy

- **Push:** A container image is meticulously crafted from the /build directory. This image is then securely uploaded to the Elastic Container Registry (ECR).

- **Deploy:** The latest container image from ECR is meticulously retrieved. This image is utilized to seamlessly update the ECS service, ensuring smooth deployment.

a. Sample `.gitlab-ci.yml` file



Untitled-2

```yaml
image: node:18-alpine

stages:
  - build
  - push
  - deploy

before_script:
  - apk add --no-cache python3 py3-pip
  - apk add --no-cache docker
  - python3 -m venv /tmp/awscli-venv
  - source /tmp/awscli-venv/bin/activate
  - pip install awscli
  - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY

build:
  stage: build
  script:
    - yarn install
    - CI=false yarn build
  artifacts:
    paths:
      - build/

push:
  stage: push
  dependencies:
    - build
  image: docker:latest
  services:
    - docker:dind
  script:
    # Login to ECR using AWS CLI
    - aws ecr-public get-login-password --region us-east-1 | docker login --username AWS --password-stdin public.ecr.aws/********

    # Build and push Docker image
    - echo -e "FROM nginx:alpine\nCOPY build/ /usr/share/nginx/html" > Dockerfile
    - echo -e "server {\\
      \n\tlisten 80;\\
      \n\tlocation / {\\
      \n\t\troot /usr/share/nginx/html;\\
      \n\t\tindex index.html index.htm;\\
      \n\t\ttry_files \$uri \$uri/ /index.html;\\
      \n\t}\\
      \n\terror_page 404 =200 /index.html;\\
      \n}" > default.conf
    - echo -e "COPY default.conf /etc/nginx/conf.d/default.conf" >> Dockerfile
    - docker build -t fa-mock:$CI_COMMIT_SHA .
    - docker tag fa-mock:$CI_COMMIT_SHA public.ecr.aws/********/fa-mock:$CI_COMMIT_SHA
    - docker push public.ecr.aws/********/fa-mock:$CI_COMMIT_SHA

deploy:
  stage: deploy
  script:
    - aws ecs update-service \
      --cluster $ECS_CLUSTER_NAME \
      --service $ECS_SERVICE_NAME \
      --task-definition $ECS_TASK_DEFINITION_ARN \
      --desired-count 1 \
      --force-new-deployment

  only:
    - dev-branch
```

b. Explanation

This .gitlab-ci.yml file acts as the blueprint for your project's continuous integration and continuous delivery (CI/CD) pipeline. Let's delve into its workings:

Setting the Stage:

- Node.js: The pipeline utilizes the node:18-alpine image as its foundation, providing a Node.js environment for your application.
- Stage by Stage: The pipeline is divided into three distinct stages: build, push, and deploy, each handling crucial tasks.

Before the Curtain Rises:

- Installing necessary package: The script prepares the environment by installing necessary tools like Python 3, pip, and Docker. It also sets up a virtual environment for awscli, the AWS command-line tool.
- Connecting to AWS: awscli is installed and configured to interact with your AWS account using secure environment variables, ensuring your credentials remain confidential.

Building the source:

- Gathering the Tools: The build stage uses yarn to install all the dependencies your application needs to function.
- Crafting the Application: The yarn build command assembles your application, likely with optimizations enabled thanks to the **`CI=false`** flag.
- Storing the Assets: The output of the build process (located in the build/ directory) is saved and made available for subsequent stages.

Pushing the Image:

- Leveraging Dependencies: This stage depends on the output from the build stage, ensuring the built application is available.
- Docker: The docker:latest image is used within the docker:dind service, enabling Docker commands inside the container.
- Connecting to ECR: The script utilizes awscli to securely log in to your Elastic Container Registry (ECR) using environment variables.
- Building and Tagging: A simple Dockerfile is created, incorporating Nginx and the build artifacts. A basic Nginx configuration file is also generated. Finally, the Docker image is built, tagged with the commit SHA for identification, and pushed to your ECR repository.

Deploying to ECS:

- Updating the ECS Service: This stage uses the **`aws ecs update-service`** command to update your ECS service with the latest task definition based on the built and pushed image.

- Selective Deployment: The only directive ensures this stage only runs when you push changes to the dev-branch, preventing unnecessary deployments.

2. Using CI/CD Variables for Secure and Efficient Pipelines

a. Why using CI/CD variables

CI/CD variables are a powerful tool for managing configuration and secrets in your GitLab pipelines. They allow you to:

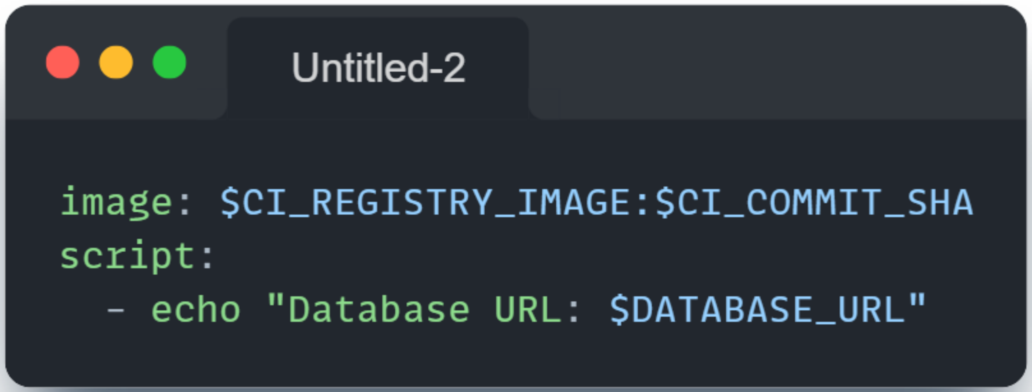
- **Control pipeline behavior**: Set environment-specific settings, trigger different jobs depending on conditions, etc.
- **Store reusable values**: Avoid hardcoding sensitive information like API keys or database credentials in your `.gitlab-ci.yml` file.
- **Improve security**: Use protected variables and masking to keep sensitive data safe.

b. Defining and Using Variables

To add or update variables in the project settings:
Step 1: Go to your project's **Settings** > **CI/CD** > **Variables**.
Step 2: Click **Add variable** and fill in the details:

- Key: a unique name (e.g., DATABASE_URL).
- Value: the actual data (e.g., the database connection string).
- Type: "Variable" (default) or "File" for storing large amounts of data.
- Environment scope: (optional) specify which environments the variable is available in (e.g., "production").
- Protected variable: (optional) restrict access to protected branches/tags only.
- Mask variable: (optional) hide the value in job logs.

Step 3: Use the variable in your `.gitlab-ci.yml` file or job scripts:



```
image: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
script:
  - echo "Database URL: $DATABASE_URL"
```

For more information about setting and using CI/CD variables, see document at this link https://docs.gitlab.com/ee/ci/variables

3. Control Your CI/CD with Protected Branches and Tags (optional)

If you want to control your CI/CD pipeline and prevent unnecessary deployments from all feature branches, you can set up protected branch rules or protected tags.

a. Understand Protected branches and Protected tags

Protected branches and tags are powerful tools for managing your GitLab repository and CI/CD pipelines. They allow you to**:**

- **Protected branches:** with a protected branch, you can control which users can merge, push, force push into the branch, which user can unprotect the branch. In CI/CD, you can use protected branches to restrict automated deployments triggered by pushes to protected branches.
- **Protected tags:** is used to safeguard specific tags from being overwritten or deleted unintentionally. In CI/CD, it is used to control which tags can trigger deployments.

b. Settings

To protect a branch:

Step 1: On the left sidebar, select **Search** or go to and find your project.

Step 2: Select **Settings** > **Repository**.

Step 3: Expand **Protected branches**.

Step 4: Select **Add protected branch**.

Step 5: From the **Branch** dropdown list, select the branch you want to protect.

Step 6: From the **Allowed** to merge list, select a role that can merge into this branch.

Step 7: From the **Allowed** to push and merge list, select a role that can push to this branch.

In GitLab Premium and Ultimate, you can also add groups or individual users to **Allowed to merge** and **Allowed to push and merge**.

Step 8: Select **Protect**.

To protect with a tag:

Step 1: On the left sidebar, select **Search** or go to and find your project.

Step 2: Select **Settings** > **Repository**.

Step 3: Expand **Protected tags**.

Step 4: Select **Add new**.

Step 5: To protect a single tag, select **Tag**, then choose your tag from the dropdown list.

Step 6: To protect all tags with names matching a string:

- Select **Tag**.
- Enter the string to use for tag matching. Wildcards (*) are supported.
- Select **Create wildcard**.

Step 7: In **Allowed** to create, select roles that may create protected tags.

In GitLab Premium and Ultimate, you can also add groups or individual users to **Allowed to create**.

Step 8: Select **Protect**.

4. Run Your CI/CD Pipelines on AWS with GitLab Runner

a. What is GitLab Runner?

Think of GitLab Runner as a robot that follows your instructions in your GitLab CI/CD pipelines. It can run tests, build your application, and deploy it to production, all automatically.

Notably, GitLab Runner boasts remarkable flexibility. Whether you prefer the control of your own server, the efficiency of Docker containers, or the scalable power of Kubernetes clusters, GitLab Runner readily adapts to your chosen environment. Furthermore, its wide compatibility with operating systems like Windows, macOS, and Linux ensures a smooth integration with your existing infrastructure.

In this article, we'll delve into the practical deployment of GitLab Runner on an EC2 instance running Linux, providing you with a concrete example of its implementation. Stay tuned for an in-depth exploration!

b. Setting Up GitLab Runner on EC2

Create GitLab Runner on GitLab

Step 1: Go to your project in GitLab and open the **Settings** tab.

Step 2: Under the **CI/CD** section, expand the **Runners** section.

Step 3: Click **New project runner**.

Step 4: Select your operating system where you installed the GitLab Runner. Then enter details:

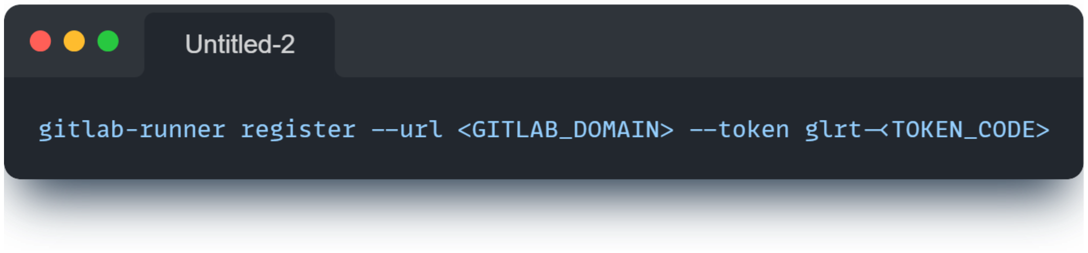
- Tags: you can give a tag here, then later you can set to run CI/CD pipeline on only tags you specified.
- Description (optional): Add a note for future reference.

- Configuration: Establish rules such as preventing the runner from accepting new jobs, allowing only the protected branch to run the CI/CD pipeline, restricting the use of the current runner to the current project only, and setting a maximum job timeout.

Step 5: Click **Create runner**: This will create a runner for your project.

Register a GitLab Runner on EC2 instance

Once you complete creating GitLab runner, you can get the GitLab runner token. Please note this token, then you will use this token when you setting your EC2 instance. Here is how we using that token later:



```
gitlab-runner register --url <GITLAB_DOMAIN> --token glrt-<TOKEN_CODE>
```

Launch EC2 instance

Step 1: Login to your AWS account and navigate to the **EC2** service console.

Step 2: Choose an **Amazon Machine Image (AMI)**. This pre-configured software template determines your operating system and basic software stack.

Step 3: Select an **Instance Type**. This defines the hardware resources allocated to your instance, like CPU, memory, and storage.

Step 4: Configure **Instance Details**. Specify the number of instances, network settings, security groups, and storage options. Create a new key pair (securely store the private key) for accessing your instance.

Step 5: Add **Storage** (Optional). Consider attaching additional storage volumes for persistent data storage.

Step 6: **Review and Launch**: Double-check your configuration and confirm any applicable charges.

Step 7: Click **Launch** to spin up your EC2 instance.

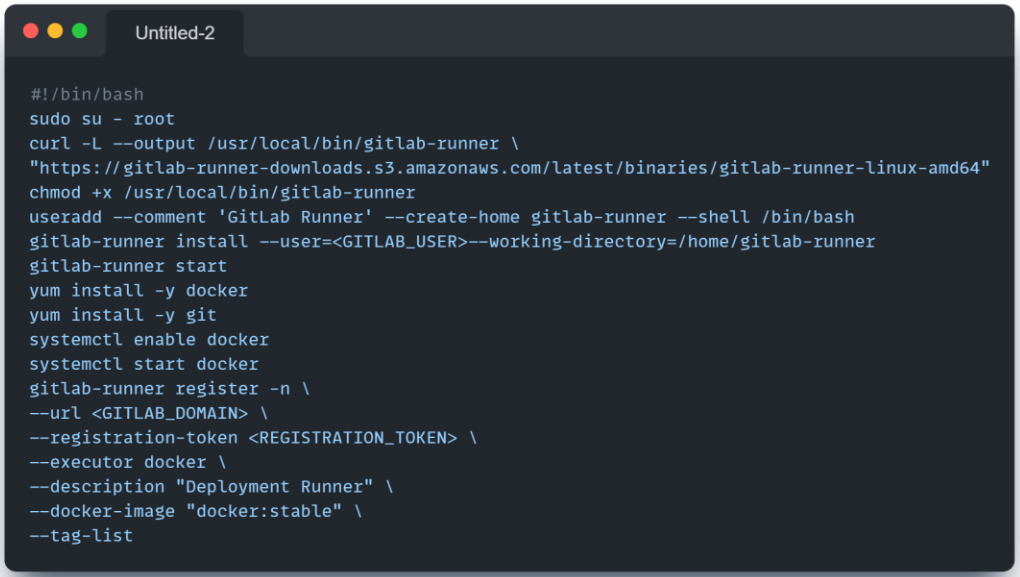
Connect EC2 with Gitlab Runner

When launching your EC2 instance, you have two options for setting up GitLab Runner:

User Data: (Automated Setup) Leverage the User Data section during launch to execute a script that automatically configures GitLab Runner on your instance. This approach offers a streamlined and hands-off experience.

SSH: (Manual Control) Connect to your running instance via SSH and execute the setup script manually. This option provides more direct control and customization, allowing you to tailor the Runner configuration to your specific needs.

Sample of connect GitLab Runner with EC2 by using User Data script



```
#!/bin/bash
sudo su - root
curl -L --output /usr/local/bin/gitlab-runner \
"https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64"
chmod +x /usr/local/bin/gitlab-runner
useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
gitlab-runner install --user=<GITLAB_USER>--working-directory=/home/gitlab-runner
gitlab-runner start
yum install -y docker
yum install -y git
systemctl enable docker
systemctl start docker
gitlab-runner register -n \
--url <GITLAB_DOMAIN> \
--registration-token <REGISTRATION_TOKEN> \
--executor docker \
--description "Deployment Runner" \
--docker-image "docker:stable" \
--tag-list
```

This script will install GitLab Runner and Docker on EC2, then connect GitLab Runner with GitLab via **`gitlab-runner register`** command.

Please replace **`GITLAB_USER`**, **`GITLAB_DOMAIN`**, **`REGISTRATION_TOKEN`** with your own values:

GITLAB_USER: Name of your GitLab User account.

GITLAB_DOMAIN: by default, it is **gitlab.com**, or if you are using GitLab organization, it will be domain of your GitLab organization (ex: git.<organization>.<top-domain>).

REGISTRATION_TOKEN: the token that you get when you create GitLab Runner ealier.

5. Create a Secure Home for Your Docker Images with Amazon ECR

Amazon ECR is a container image registry specifically designed for storing and managing Docker images used in your applications. This section will guide you through creating a secure and efficient ECR repository using the AWS Management Console:

Step 1: Access the **ECR** console at https://console.aws.amazon.com/ecr/repositories and choose your desired region.

Step 2: Click **Create repository** and enter a unique name.

Step 3: Customize settings (optional):

- **Tag immutability**: Prevent accidental overwrites by enabling this option.
- **Image scanning**: Consider configuring scanning at the registry level for more flexibility.
- **KMS encryption**: Choose to encrypt images for enhanced security.

Step 4: Review and create: Double-check your settings and

Step 5: Click **Create repository** to finalize.

6. Deploy Your Website on AWS ECS

This guide will show you how to create an ECS cluster and service to host your website on AWS. While it requires some technical knowledge, we'll break down the steps into manageable parts.

a. Creating the Blueprint: Task Definition

A task definition is a blueprint for your application. It is a text file in JSON format that describes the parameters and one or more containers that form your application.

Here is how to create a Task definition to deploy your website:

Step 1: Login to your AWS Management Console, go to **ECS** dashboard at this link https://console.aws.amazon.com/ecs/v2.

Step 2: Go to **Task Definitions**.

Step 3: Click **Create new Task Definition** and give it a unique name.

Step 4: Choose your **Launch type** (in this article, I mention about using Fargate so I choose Fargate), **container image** (e.g., from ECR), ports, and other settings.

Step 5: **Review and create** your task definition.

b. Create ECS Cluster and Service

Building the Foundation: ECS Cluster

Step 1: Login to your AWS Management Console, go to **ECS** dashboard at this link https://console.aws.amazon.com/ecs/v2.

Step 2: Choose **Fargate** for a pay-as-you-go option and give it a name.

Cluster name: Name can be a maximum of 255 characters. The valid characters are letters (uppercase and lowercase), numbers, hyphens, and underscores.

Infrastructure: there are 3 options for you: ***Fargate****,* ***EC2****, or* ***external instance*** *using ECS. In this article, I will use Fargate. With Fargate, you will pay as you go, if you have tiny, batch, or burst workloads or for zero maintenance overhead think about Fargate. The cluster has Fargate and Fargate Spot capacity providers by default.*

Step 3: **Review and create** your cluster.

Create ECS Service

Step 1: Open the console at https://console.aws.amazon.com/ecs/v2.

Step 2: In the navigation page, choose **Clusters**.

Step 3: On the **Clusters** page, choose the cluster to create the service in.

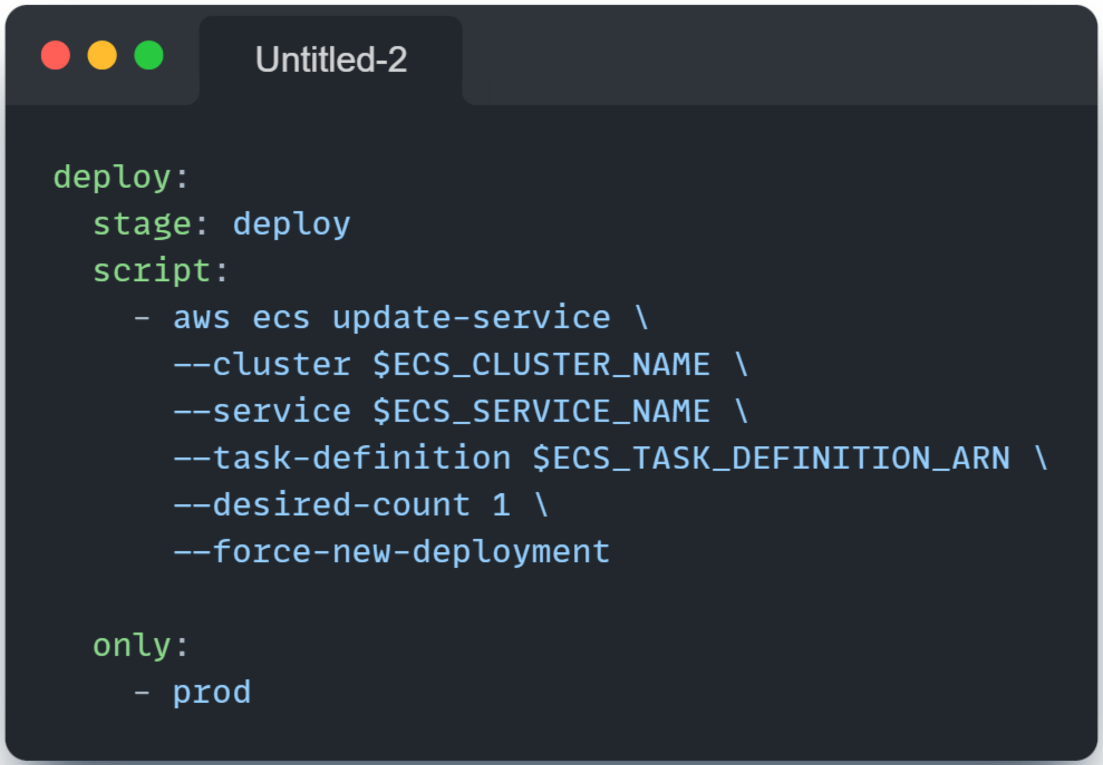
Step 4: From the **Services** tab, choose **Create**.

Step 5: Under **Deployment configuration**, specify how your application is deployed.

- For **Application type**, choose Service.
- For **Task definition**, choose the task definition family and revision to use.
- For **Service name**, enter a name for your service.
- For **Desired tasks**, enter the number of tasks to launch and maintain in the service.

Step 6: **Review and create** your service.

c. Automating Deployment with GitLab Runner


Untitled-2
```
deploy:
  stage: deploy
  script:
    - aws ecs update-service \
      --cluster $ECS_CLUSTER_NAME \
      --service $ECS_SERVICE_NAME \
      --task-definition $ECS_TASK_DEFINITION_ARN \
      --desired-count 1 \
      --force-new-deployment

  only:
    - prod
```

When GitLab Runner completes tasks on build and push image to ECR, then it will trigger actions defined in deploy stage.

At the deploy stage, it will use **`aws ecs update-service`** command to update ECS service with pre-defined ECS task definition.

Please defined these variables under **Settings** > **CI/CD** > **Variables**:

`$ECS_CLUSTER_NAME`: is the name of ECS cluster that you created earlier.

`$ECS_SERVICE_NAME`: is the name of a service that you created earlier.

`$ECS_TASK_DEFINITION_ARN`: is ARN of task definition you have defined

`--desired-count <number>`: where number is number of tasks will be run after updated.

`--force-new-deployment`: using this parameter to let ECS always update when newer version of image is pushed to ECR.

7. Advanced deployment task on Fargate with load balancer and domain registration

a. Create service load balancing for ECS service

Overview

Your Amazon ECS service can optionally be configured to use Elastic Load Balancing to distribute traffic evenly across the tasks in your service.

Amazon ECS services hosted on AWS Fargate support the Application Load Balancer and Network Load Balancer load balancer types.

- Application Load Balancers are used to route HTTP/HTTPS (or layer 7) traffic.
- Network Load Balancers are used to route TCP or UDP (or layer 4) traffic.

Application Load Balancers offer several features that make them attractive for use with Amazon ECS services:

- Each service can serve traffic from multiple load balancers and expose multiple load balanced ports by specifying multiple target groups.
- They are supported by tasks hosted on both Fargate and EC2 instances.
- Application Load Balancers allow containers to use dynamic host port mapping (so that multiple tasks from the same service are allowed per container instance).
- Application Load Balancers support path-based routing and priority rules (so that multiple services can use the same listener port on a single Application Load Balancer).

Config target group for routing

In this section, you create a target group for your load balancer and the health check criteria for targets that are registered within that group.

Each target group is used to route requests to one or more registered targets. When a rule condition is met, traffic is forwarded to the corresponding target group.

Your load balancer distributes traffic between the targets that are registered to its target groups. When you associate a target group to an Amazon ECS service, Amazon ECS automatically registers and deregisters containers with your target group. Because Amazon ECS handles target registration, you do not add targets to your target group at this time.

Create targe group:

Step 1: Open the Amazon **EC2** console at https://console.aws.amazon.com/ec2.

Step 2: On the navigation pane, under **Load Balancing**, choose **Target Groups**.

Step 3: Choose **Create target group**.

Step 4: For Choose a target type, Instances to register targets by instance ID, IP addresses to register targets by IP address, or Lambda function to register a Lambda function as a target.

Important

If your service's task definition uses the awsvpc network mode (which is required for the Fargate launch type), you must choose IP addresses as the target type This is because tasks that use the awsvpc network mode are associated with an elastic network interface, not an Amazon EC2 instance.

Step 5: For **Target group name**, enter a name for the target group. This name must be unique per region per account, can have a maximum of 32 characters, must contain only alphanumeric characters or hyphens, and must not begin or end with a hyphen.

Step 6: (Optional) For **Protocol** and **Port**, modify the default values as needed.

Step 8: If the target type is **IP addresses**, choose **IPv4** as the **IP address type**, otherwise skip to the next step.

Note that only targets that have the selected IP address type can be included in this target group. The IP address type cannot be changed after the target group is created.

Step 9: For **VPC**, select a virtual private cloud (VPC). Note that for **IP addresses** target types, the VPCs available for selection are those that support the **IP address type** that you chose in the previous step.

Step 10: (Optional) For **Protocol version**, modify the default value as needed.

Step 11: (Optional) In the **Health checks** section, modify the default settings as needed.

Step 11: If the target type is **Lambda function**, you can enable health checks by selecting **Enable** in the **Health checks** section.

Step 12: (Optional) Add one or more tags as follows:

- Expand the **Tags** section.
- Choose **Add tag**.
- Enter the tag key and the tag value.

Step 13: Choose **Next**.

Step 14: Choose **Create target group**.

Define your load balancer

First, provide some basic configuration information for your load balancer, such as a name, a network, and a listener.

A listener is a process that checks for connection requests. It is configured with a protocol and a port for the frontend (client to load balancer) connections, and protocol and a port for the backend (load balancer to backend instance) connections. In this example, you configure a listener that accepts HTTP requests on port 80 and sends them to the containers in your tasks on port 80 using HTTP.

To configure your load balancer and listener:

Step 1: Open the Amazon EC2 console at https://console.aws.amazon.com/ec2.

Step 2: In the navigation pane, under **Load Balancing**, choose **Load Balancers**.

Step 3: Choose **Create Load Balancer**.

Step 4: Under **Application Load Balancer**, choose **Create**.

Step 5: Under **Basic configuration**, do the following:

- For **Load balancer name**, enter a name for your load balancer. For example, `my-nlb`. The name of your Application Load Balancer must be unique within your set of Application Load Balancers and Network Load Balancers for the Region. Names can have a maximum of 32 characters and can contain only alphanumeric characters and hyphens. They cannot begin or end with a hyphen, or with **`internal-`**.
- For **Scheme**, choose **Internet-facing** or **Internal**. An internet-facing load balancer routes requests from clients to targets over the internet. An internal load balancer routes requests to targets using private IP addresses.
- For **IP address type**, choose the IP addressing for the container's subnets.

Step 6: Under **Network mapping**, do the following:

- For **VPC**, select the same VPC that you used for the container instances on which you intend to run your service.
- For **Mappings**, select the Availability Zones to use for your load balancer. If there is one subnet for that Availability Zone, it is selected. If there is more than one subnet for that Availability Zone, select one of the subnets. You can select only one subnet per Availability Zone. Your load balancer subnet configuration must include all Availability Zones that your container instances reside in.

Step 7: Under **Security groups**, do the following:

For **Security groups**, select an existing security group, or create a new one.

The security group for your load balancer must allow it to communicate with registered targets on both the listener port and the health check port. The console can create a security group for your load balancer on your behalf with rules that allow this communication. You can also create a security group and select it instead.

(Optional) To create a new security group for your load balancer, choose Create a new security group.

Step 8: Under **Listeners and routing**, do the following:

The default listener accepts HTTP traffic on port 80. You can keep the default protocol and port. For **Default action**, choose the target group that you created.

You can optionally add an HTTPS listener after you create the load balancer.

Step 9: (Optional) You can use **Add-on services**, such as the **AWS Global Accelerator** to create an accelerator and associate the load balancer with the accelerator.

The accelerator name can have up to 64 characters. Allowed characters are a-z, A-Z, 0-9, . and - (hyphen). After the accelerator is created, you can use the AWS Global Accelerator console to manage it.

Step 10: (Optional) Tag your Application Load Balancer. Under **Tag and create**, do the following

- Expand the **Tags** section.
- Choose **Add tag**.
- Enter the tag key and the tag value.

Step 11: Review your configuration and choose **Create load balancer**.

b. Configuring Amazon Route53 to route traffic to an Application Load Balancer

To route traffic to an ELB load balancer

Step 1: If you created the Route 53 hosted zone and ELB load balancer using the same account, skip to. If you created the hosted zone and the ELB load balancer using different accounts, perform the procedure Getting the DNS name for an Elastic Load Balancing load balancer to get the DNS name for the load balancer.

Step 2: Sign in to the AWS Management Console and open the **Route 53** console at https://console.aws.amazon.com/route53.

Step 3: In the navigation pane, choose **Hosted zones**.

Step 4: Choose the name of the hosted zone that has the domain name that you want to use to route traffic to your load balancer.

Step 5: Choose **Create record**.

Step 6: Specify the following values:

- **Routing policy**: Choose the applicable routing policy. For more information, see Choosing a routing policy.
- **Record name**: Enter the domain or subdomain name that you want to use to route traffic to your ELB load balancer. The default value is the name of the hosted zone. For example, if the name of the hosted zone is example.com and you want to use acme.example.com to route traffic to your load balancer, enter **acme**.

- **Alias**: If you are using the **Quick create** record creation method, turn on **Alias**.
- **Value/Route traffic to**: Choose Alias to **Application and Classic Load Balancer** or **Alias to Network Load Balancer**, then choose the Region that the endpoint is from. If you created the hosted zone and the ELB load balancer using the same AWS account, choose the name that you assigned to the load balancer when you created it. If you created the hosted zone and the ELB load balancer using different accounts, enter the value that you got in step 1 of this procedure.
- **Record type**: Choose **A – IPv4 address**.
- **Evaluate target health**: If you want Route 53 to route traffic based on the health of your resources, choose **Yes**. For more information about checking the health of your resources, see Creating Amazon Route 53 health checks and configuring DNS failover.

Step 7: Choose **Create records**. Changes generally propagate to all Route 53 servers within 60 seconds. When propagation is done, you'll be able to route traffic to your load balancer by using the name of the alias record that you created in this procedure.

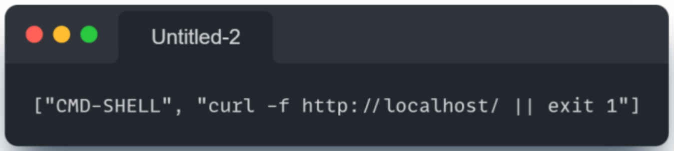
V. Troubleshoot issues

1. Health Check Service

a. Define service's health check

You can define service health check when creating **Task definition**. Here is what you can use:

- **Command**: This is a string array representing the command that the container runs to determine if it is healthy. An exist code with 0 is success, non-zero indicates failure.



```
["CMD-SHELL", "curl -f http://localhost/ || exit 1"]
```

- **Interval**: This is the period in seconds between each health check execution. You may specify between 5 and 300 seconds (about 5 minutes). The default value is 30 seconds.
- **Retries**: The number of times to retry the health check before considering the container as `unhealthy`.
- **Timeout**: This is the period in seconds to wait for a health check to succeed before it is considered a failure.

b. What will you do when the service's health check fail?

Check the Task Logs: The first step is to check the logs of the failing task. You can access these logs if you have configured your tasks to send log information to Amazon CloudWatch Logs.

Inspect the Task: Use the `DescribeTasks` API operation to get more information about the tasks. This can provide useful information about why the task has failed its health check.

Check the Task Definition: Make sure that the task definition is correctly configured. This includes checking the Docker image, the command that is being run, and any environment variables.

Check the Service Events: In the Amazon ECS console, you can check the service events for your service. These events can provide information about why tasks are not running or why they are failing health checks.

Check the Container Instance: If the task is failing on a specific container instance, there might be an issue with that instance. Check the instance's CPU and memory usage, and make sure it has enough resources to run the task.

Redeploy the Service: If you have made changes to the task definition or the service configuration, you can redeploy the service. This will stop the current running tasks and start new ones with the updated configuration.

2. Rollback Plan

a. Overview

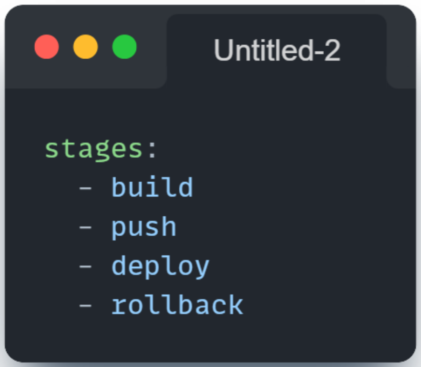
If the deployment passes but business requirements fail, it means the new version of the application is not meeting the expected functionality or performance. This could be due to a variety of reasons such as bugs in the code, issues with new features, performance problems, etc.

To rollback in this scenario, you would need to identify the previous Docker image in ECR and update the ECS service to use that image.

The solution is you could tag each image with the Git commit SHA and a `latest` tag. The `latest` tag should point to the most recent stable version of the application. If you need to rollback, you can update ECS service to use the latest image.

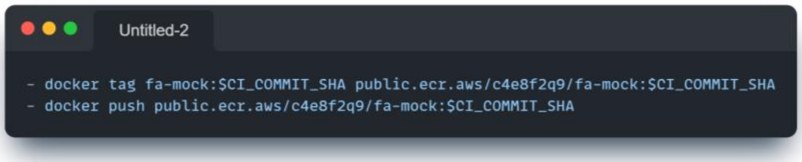
b. Update gitlab-ci.yml file for rollback

Add rollback in stages:



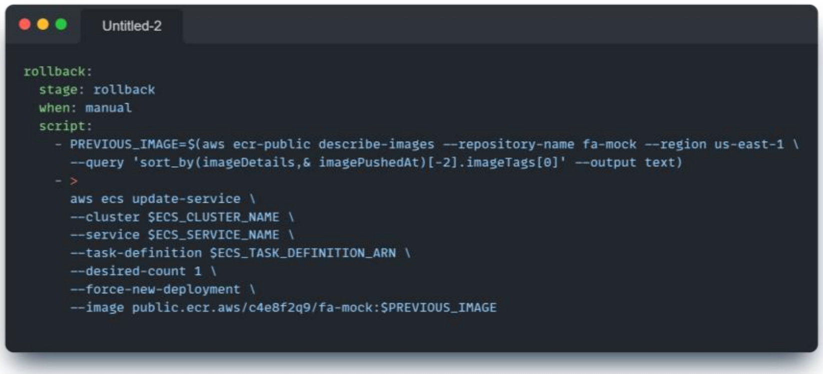
```
stages:
  - build
  - push
  - deploy
  - rollback
```

Update docker build command: to using `$CI_COMMIT_SHA` as tag name of image



```
- docker tag fa-mock:$CI_COMMIT_SHA public.ecr.aws/c4e8f2q9/fa-mock:$CI_COMMIT_SHA
- docker push public.ecr.aws/c4e8f2q9/fa-mock:$CI_COMMIT_SHA
```

Define rollback stage:



```
rollback:
  stage: rollback
  when: manual
  script:
    - PREVIOUS_IMAGE=$(aws ecr-public describe-images --repository-name fa-mock --region us-east-1 \
      --query 'sort_by(imageDetails,& imagePushedAt)[-2].imageTags[0]' --output text)
    - >
      aws ecs update-service \
      --cluster $ECS_CLUSTER_NAME \
      --service $ECS_SERVICE_NAME \
      --task-definition $ECS_TASK_DEFINITION_ARN \
      --desired-count 1 \
      --force-new-deployment \
      --image public.ecr.aws/c4e8f2q9/fa-mock:$PREVIOUS_IMAGE
```

Manually trigger the rollback stage:

In GitLab CI/CD, stages that are set to `when: manual` can be triggered manually from the GitLab interface. Here's how you can do it:

Step 1: Go to your project in GitLab.

Step 2: Navigate to **CI/CD** > **Pipelines**.

Step 3: You'll see a list of pipelines. Click on the pipeline that you want to trigger the rollback for.

Step 4: In the pipeline details, you'll see a play button next to the rollback stage. Click on this button to manually trigger the rollback.

VI. Conclusion

This article has provided a comprehensive overview of building a streamlined CI/CD pipeline for deploying your ReactJS website using GitLab and AWS services. By implementing this approach, you gain significant benefits, including:

- **Reduced deployment time and effort:** Automate deployments and free your team to focus on more strategic tasks.
- **Improved software quality:** Catch and rectify issues through automated testing, ensuring a robust and secure website.
- **Faster response to changes:** Respond swiftly to user feedback and security threats with rapid deployments.
- **Enhanced developer productivity:** Eliminate manual deployments and allow developers to focus on building new features and improvements.

Remember that your specific implementation may vary based on your project needs and the complexity of your website. However, this article provides a solid foundation for building a robust and efficient CI/CD pipeline for your ReactJS website, empowering you to achieve faster and more reliable deployments.

VII. References

1. AWS Documents

[1] Tutorial: Get started with Amazon EC2 Linux instances - Amazon Elastic Compute Cloud. (n.d.). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html

[2] Set up to use Amazon ECS - Amazon Elastic Container Service. (n.d.). https://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html

[3] Service load balancing - Amazon Elastic Container Service. (n.d.). https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-load-balancing.html

[4] Target groups for your Application Load Balancers - Elastic Load Balancing. (n.d.). https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html

[5] AWS::ECS::TaskDefinition HealthCheck - AWS CloudFormation. (n.d.). https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-ecs-taskdefinition-healthcheck.html

2. GitLab Documents

[6] GitLab Runner | GitLab. (n.d.). https://docs.gitlab.com/runner

[7] GitLab CI/CD variables | GitLab. (n.d.). https://docs.gitlab.com/ee/ci/variables

[8] Tutorial: Create and run your first GitLab CI/CD pipeline | GitLab. (n.d.-b). https://docs.gitlab.com/ee/ci/quick_start