Certainly! Below is the **complete project documentation** including **Task 1**, structured in Markdown format. You can easily convert this Markdown file to PDF using tools like **Pandoc**, **Markdown editors**, or **online converters**.

Project Documentation

Table of Contents

1. Project Overview
2. Task 1: Setting Up the Development Environment
3. Task 2: Setting Up AWS Infrastructure
    - Subtask 3: Configuring AWS ALB and Target Groups
    - Subtask 4: Setting Up Amazon ECR and IAM Roles
4. Task 3: Dockerizing the Application and CI/CD Pipeline
    - Subtask 1: Containerizing the Movies App
    - Subtask 2: Creating Jenkins Pipeline to Build and Push Docker Images
5. Troubleshooting and Resolutions
    - Issue: Unable to Locate Credentials
    - Issue: SSH Authentication Errors
    - Issue: Docker Daemon Permission Denied
    - Issue: Jenkins Pipeline Errors
6. Conclusion and Next Steps

Project Overview

This project involves setting up a robust CI/CD pipeline using **Jenkins**, **Docker**, **AWS Elastic Container Registry (ECR)**, and **Terraform** to deploy a containerized Node.js application. The primary objectives include:

1. **Development Environment Setup**: Preparing the local development environment for building and testing the application.
2. **Infrastructure Setup**: Configuring AWS resources such as Application Load Balancers (ALB), Target Groups, and ECR repositories using Terraform.
3. **Application Containerization**: Dockerizing a sample Node.js application (Movies App) to prepare it for deployment.
4. **CI/CD Pipeline**: Implementing a Jenkins pipeline that automates the build, test, and deployment processes, including pushing Docker images to ECR and deploying them to an application server.

Task 1: Setting Up the Development Environment

Objective

Prepare your local development environment to develop, build, and test the Movies App. This includes installing necessary tools, cloning the repository, and ensuring the application runs correctly before containerization.

Steps

1. **Install Required Software**:

   Ensure the following software is installed on your **local development machine**:

   - **Node.js and npm**: Required for running the Node.js application.
     - **Installation**:
       - **macOS**:

         ```
         brew install node
         ```

       - **Ubuntu/Debian**:

         ```
         sudo apt-get update
         sudo apt-get install -y nodejs npm
         ```

       - **Windows**:
         - Download and install from Node.js Official Website.
   - **Git**: For version control and cloning repositories.
     - **Installation**:
       - **macOS**:

         ```
         brew install git
         ```

       - **Ubuntu/Debian**:

```
sudo apt-get update
sudo apt-get install -y git
```

        - **Windows**:
            - Download and install from Git Official Website.
    - **Docker**: To build and run Docker containers.
        - **Installation**:
            - Follow the official Docker installation guide for your operating system: Docker Installation.
    - **Terraform**: For infrastructure as code.
        - **Installation**:
            - Download and install from Terraform Downloads.

2. **Clone the Movies App Repository**:

   Where to Run: On your **local development machine**.

   ```
   git clone https://github.com/samaronybarros/movies-app.git
   cd movies-app
   ```

3. **Install Application Dependencies**:

   Where to Run: Inside the `movies-app` directory.

   ```
   npm install
   ```

4. **Run the Application Locally**:

   Where to Run: Inside the `movies-app` directory.

   ```
   npm start
   ```

    - The application typically runs on **port 3000**.
    - Open a browser and navigate to http://localhost:3000 to verify the application is running correctly.

5. **Verify Application Functionality**:
    - Ensure that the Movies App interface loads without errors.
    - Test basic functionalities like viewing movies, adding new movies, etc., to confirm everything works as expected.

6. **Prepare for Dockerization**:
    - Ensure that all functionalities work correctly in the local environment before proceeding to containerize the application.
    - Make any necessary code adjustments or optimizations based on local testing.

Where to Run: All steps are executed on your **local development machine**.

Task 2: Setting Up AWS Infrastructure

Subtask 3: Configuring AWS ALB and Target Groups

Objective: Set up an internet-facing Application Load Balancer (ALB) to route traffic to the Jenkins and Application servers securely.

Steps:

1. **Create an Application Load Balancer (ALB)**:
    - Define the ALB using Terraform with the necessary configurations such as listeners, security groups, and subnets.
    - Example Terraform code provided in previous interactions.

2. **Create Target Groups**:
    - Set up separate target groups for Jenkins and the Application server.
    - Configure health checks and specify the protocol and port.

3. **Configure Listeners and Rules**:
    - Set up listeners on the ALB to forward traffic to the appropriate target groups based on path patterns.

4. **Security Group Considerations**:
    - Ensure ALB security groups allow inbound HTTP traffic on port 80 from the internet.
    - Configure the Jenkins and Application server security groups to allow traffic from the ALB.

5. **Testing**:
    - Access the Jenkins and Application servers via the ALB's DNS name to verify proper routing and accessibility.

Where to Run: All Terraform configurations should be executed from your **local development machine** or a dedicated CI/CD server with Terraform installed.

Subtask 4: Setting Up Amazon ECR and IAM Roles

Objective: Create an Amazon ECR repository for Docker images, configure IAM roles for Jenkins and Application servers, and establish SSH connectivity between Jenkins and the Application server.

Steps:

1. **Create an Amazon ECR Repository**:
   - Use the AWS Management Console or AWS CLI to create a private ECR repository named `node-app`.
   - Note the repository URI for future reference.
2. **Set Up IAM Roles**:
   - **For Jenkins Server (`10.0.1.110`)**:
     - Create an IAM role (`JenkinsECRRole`) with permissions to push/pull from ECR.
     - Attach the role to the Jenkins EC2 instance via the AWS EC2 Console.
   - **For Application Server (`10.0.2.43`)**:
     - Create an IAM role (`AppECRRole`) with read-only access to ECR.
     - Attach the role to the Application EC2 instance.
3. **Authenticate Jenkins and Application Servers to ECR**:
   - Utilize the attached IAM roles to allow Docker commands on both servers to interact with ECR without manual credential configuration.
4. **Generate SSH Key Pair for Jenkins → Application Server**:
   - On the Jenkins server (`10.0.1.110`), generate an SSH key pair (`jenkins_app_rsa`).
   - Copy the public key to the Application server (`10.0.2.43`) by appending it to the `~/.ssh/authorized_keys` file.
5. **Test SSH Connectivity**:
   - Create a sample Jenkins job that uses the SSH credentials to connect to the Application server and execute basic commands, ensuring the SSH setup works correctly.

Where to Run:

- **AWS Console/AWS CLI**: For creating ECR repositories and IAM roles.
- **Jenkins Server (`10.0.1.110`)**: For generating SSH keys and configuring SSH access.
- **Application Server (`10.0.2.43`)**: For setting up `authorized_keys`.

Task 3: Dockerizing the Application and CI/CD Pipeline

Subtask 1: Containerizing the Movies App

Objective: Dockerize the `movies-app` Node.js application, test it locally, and push the code along with the Dockerfile to a private GitHub repository.

Steps:

1. **Create a Private GitHub Repository**:
   - Name it `movies-app-docker`.
   - Ensure it is set to **Private**.
   - Clone the repository to your **local development machine**.
2. **Clone the Movies App Locally**:
   - Clone the sample application repository:

     ```
     git clone https://github.com/samaronybarros/movies-app.git
     ```

   - Navigate to the cloned directory:

     ```
     cd movies-app
     ```

   - Verify the structure:

     ```
     movies-app/
     ├── server.js
     ├── package.json
     ├── package-lock.json
     ├── .gitignore
     └── ...
     ```

3. **Prepare the Dockerized Project**:
   - Create a new directory for Dockerization:

```
mkdir ../movies-app-docker
cp -r * ../movies-app-docker/
cd ../movies-app-docker
```

- Initialize a new Git repository if not already initialized:

```
git init
```

- Add a `.gitignore` file to exclude `node_modules`:

```
echo "node_modules/" >> .gitignore
```

4. **Write a Dockerfile**:
   - Create a `Dockerfile` with the following content:

```
# Use the Alpine version of Node
FROM node:alpine

# Create and set app directory
WORKDIR /usr/src/app

# Copy package files first
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the code
COPY . .

# Expose the port your app listens on
EXPOSE 3000

# Command to start the application
CMD ["npm", "start"]
```

5. **Build and Run the Docker Image Locally**:
   - **Build** the Docker image:

```
docker build -t movies-app:latest .
```

   - **Run** the Docker container:

```
docker run -d -p 3000:3000 movies-app:latest
```

   - **Test** the application by navigating to http://localhost:3000 in your browser to ensure it's running correctly.
6. **Commit and Push to GitHub**:
   - Stage and commit the changes:

```
git add .
git commit -m "Initial commit of movies app with Dockerfile"
```

   - Add the remote repository:

```
git remote add origin git@github.com:vsm524565/movies-app-docker.git
```

   - Push to GitHub:

```
git push -u origin master
```

   If you encounter branch naming issues, refer to the Troubleshooting Git Push section.

Where to Run: All steps are executed on your **local development machine**.

Subtask 2: Creating Jenkins Pipeline to Build and Push Docker Images

Objective: Develop a Jenkins pipeline (Jenkinsfile) that automates the process of checking out the code from GitHub, building the Docker image, and pushing it to Amazon ECR.

Steps:

1. **Ensure Prerequisites Are Met**:
    - **Jenkins** is installed and running.
    - **Docker** is installed on the Jenkins server.
    - **IAM Roles** are correctly attached to the Jenkins server with permissions to push to ECR.
    - **ECR Repository** (`node-app-repo`) exists.
2. **Create a Jenkinsfile**:
    - Inside the `movies-app-docker` directory, create a file named `Jenkinsfile` with the following content:

```
pipeline {
    agent any

    environment {
        AWS_REGION   = "us-east-1"
        ECR_URI      = "961341532593.dkr.ecr.us-east-1.amazonaws.com/node-app-repo"
        REPO_NAME    = "node-app-repo"
        IMAGE_TAG    = "${env.BUILD_NUMBER}"
    }

    stages {
        stage('Checkout') {
            steps {
                // Pull the code from your private GitHub repo via SSH
                git(
                    branch: 'main',
                    url: 'git@github.com:YourUsername/movies-app-docker.git',
                    credentialsId: '3ab0bdd3-a0e5-4f45-9779-1f834829f47f'
                )
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    // Enable BuildKit
                    sh 'export DOCKER_BUILDKIT=1'

                    // Build the Docker image
                    sh """
                    docker build -t \${REPO_NAME}:\${IMAGE_TAG} .
                    """
                }
            }
        }

        stage('Publish to ECR') {
            steps {
                script {
                    // Login to ECR
                    sh """
                    aws ecr get-login-password --region \${AWS_REGION} | docker login --username AWS --password-stdin \$
                    """

                    // Tag the image with full ECR URI
                    sh """
                    docker tag \${REPO_NAME}:\${IMAGE_TAG} \${ECR_URI}/\${REPO_NAME}:\${IMAGE_TAG}
                    """

                    // Push to ECR
                    sh """
                    docker push \${ECR_URI}/\${REPO_NAME}:\${IMAGE_TAG}
                    """
                }
            }
        }
    }

    post {
        always {
            // Optional: Clean up Docker images to save space
            sh """
            docker rmi \${REPO_NAME}:\${IMAGE_TAG} \${ECR_URI}/\${REPO_NAME}:\${IMAGE_TAG} || true
            """
        }
    }
}
```

3. **Commit and Push the Jenkinsfile**:
    - Stage and commit the Jenkinsfile:

```
 git add Jenkinsfile
git commit -m "Add Jenkinsfile for Docker build and ECR push"
```

- Push to GitHub:

  ```
  git push origin master
  ```

  If you encounter branch naming issues, refer to the Troubleshooting Git Push section.

4. **Configure Jenkins Pipeline Job**:
   - **Create a New Pipeline Job**:
     - Navigate to **Jenkins Dashboard** → **New Item** → **Pipeline** → Enter name (e.g., `docker-build-publish`) → **OK**.
   - **Configure the Pipeline**:
     - **Pipeline Section**:
       - **Definition**: Pipeline script from SCM.
       - **SCM**: Git.
       - **Repository URL**: `git@github.com:vsm524565/movies-app-docker.git` *(for SSH)* or `https://github.com/vsm524565/movies-app-docker.git` *(for HTTPS)*.
       - **Credentials**: Select the SSH key or HTTPS token with access to the private repository.
       - **Branch**: `master` or `main` depending on your repository's default branch.
       - **Script Path**: `Jenkinsfile`.
   - **Save** the Pipeline Job.

5. **Run the Jenkins Pipeline**:
   - Click **"Build Now"** on the Jenkins job.
   - **Monitor Console Output**:
     - **Checkout Stage**: Clones the repository.
     - **Build Docker Image Stage**: Builds the Docker image using Docker BuildKit.
     - **Publish to ECR Stage**: Logs into ECR, tags the image, and pushes it to the ECR repository.
   - **Verify in AWS ECR**:
     - Navigate to **AWS Management Console** → **ECR** → **Repositories** → Select `node-app-repo`.
     - Confirm that the new Docker image with the appropriate tag (`BUILD_NUMBER`) is present.

Where to Run:

- **Jenkins Web UI**: For creating and configuring the Pipeline job.
- **Local Development Machine**: For writing and committing the Jenkinsfile.
- **Jenkins Server**: For executing the pipeline stages.

Troubleshooting and Resolutions

Issue: Unable to Locate Credentials

Error Message:

```
Unable to locate credentials. You can configure credentials by running "aws configure".
Error: Cannot perform an interactive login from a non TTY device
```

Cause: The AWS CLI cannot find valid credentials and attempts to prompt for input in a non-interactive environment like Jenkins.

Solutions:

1. **Use IAM Roles (Preferred)**:
   - Ensure that the Jenkins EC2 instance has an IAM role (`JenkinsECRRole`) attached with necessary ECR permissions.
   - No need to manually configure AWS credentials; Docker commands will automatically use the role's temporary credentials.
2. **Set AWS Credentials as Environment Variables**:
   - Export `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` in Jenkins' environment.
   - **Note**: Secure these credentials using Jenkins' Credentials Store.
3. **Configure AWS CLI Non-Interactively**:
   - Write AWS credentials to `~/.aws/credentials` within the Jenkins workspace via a script.
   - Avoid using `aws configure` which requires interactive input.

Recommendation: Use IAM roles attached to the Jenkins server for seamless and secure credential management.

Issue: SSH Authentication Errors

Error Messages:

```
Load key "upgrad-test.pem": error in libcrypto
...
Permission denied (publickey).
```

Cause: SSH key file is invalid, corrupted, or Jenkins cannot access it due to permission issues.

Solutions:

1. **Verify SSH Key File**:
   - Ensure `upgrad-test.pem` is a valid private key in PEM format.
   - Check for correct headers (`-----BEGIN RSA PRIVATE KEY-----`).
2. **Fix File Permissions**:
   - Set appropriate permissions:

     ```
     chmod 400 upgrad-test.pem
     ```

   - Ensure Jenkins has read access to the key.
3. **Correct Jenkinsfile Configuration**:
   - Remove explicit `-i /home/ubuntu/upgrad-test.pem` from SSH commands.
   - Use Jenkins' SSH credentials instead to manage keys securely.
   - Example modification in Jenkinsfile:

     ```
     ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null ubuntu@10.0.2.43
     ```

4. **Ensure SSH Key Matches Authorized Keys**:
   - Confirm that the public key corresponding to `upgrad-test.pem` is present in `~/.ssh/authorized_keys` on the Application server (`10.0.2.43`).

Recommendation: Utilize Jenkins' Credentials Store to manage SSH keys and avoid referencing key files directly in scripts.

Issue: Docker Daemon Permission Denied

Error Message:

```
permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: dial unix /var/run/docker.sock
```

Cause: The Jenkins user lacks permissions to interact with the Docker daemon.

Solutions:

1. **Add Jenkins User to Docker Group**:
   - SSH into Jenkins server.
   - Add `jenkins` to the `docker` group:

     ```
     sudo usermod -aG docker jenkins
     ```

   - Restart Jenkins service:

     ```
     sudo systemctl restart jenkins
     ```

2. **Verify Group Membership**:
   - Confirm `jenkins` is part of the `docker` group:

     ```
     groups jenkins
     ```

3. **Alternative: Use `sudo` with Docker Commands**:
   - Prefix Docker commands with `sudo` in Jenkinsfile.
   - **Note**: Requires configuring `sudoers` for password-less access, which poses security risks.

Recommendation: Add Jenkins user to the Docker group to allow Docker commands without `sudo` .

Issue: Jenkins Pipeline Errors

Error Message:

```
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 16: Expected a step @ line 16, column 32.
credentialsId: '3ab0bdd3-a0e5-4f45-9779-1f834829f47f'
...
```

Cause: Incorrect syntax in the Jenkinsfile, specifically in the `git` step within the Checkout stage.

Solutions:

1. **Correct `git` Step Syntax**:
    - Enclose parameters within parentheses:

      ```
      git(
          branch: 'main',
          url: 'git@github.com:YourUsername/movies-app-docker.git',
          credentialsId: '3ab0bdd3-a0e5-4f45-9779-1f834829f47f'
      )
      ```

2. **Remove Redundant Checkout Stages**:
    - Rely on Jenkins' automatic SCM checkout at the beginning of the pipeline.
    - Remove any additional `Checkout` stages unless necessary.

3. **Ensure Consistent SCM Configuration**:
    - Use either HTTPS or SSH consistently for repository access.
    - Configure Jenkins credentials accordingly.

Recommendation: Use the corrected `git` step with proper syntax and eliminate redundant checkout stages to streamline the pipeline.

Conclusion

By completing the above tasks and subtasks, you have established a solid foundation for deploying a containerized Node.js application using Jenkins and AWS services. The key achievements include:

1. **Development Environment Setup**:
    - Installed necessary tools and dependencies.
    - Verified the application runs correctly in the local environment.

2. **AWS Infrastructure Setup**:
    - Configured ALB and Target Groups to manage application traffic.
    - Set up ECR repositories and IAM roles for secure Docker image management.

3. **Application Containerization**:
    - Successfully Dockerized the Movies App.
    - Tested Docker builds locally to ensure application integrity.

4. **CI/CD Pipeline Implementation**:
    - Developed a Jenkins pipeline that automates the build and push processes.
    - Addressed and resolved common pipeline errors to ensure smooth operations.