1.0 Overview

1.1 Problem Statement

Portland General Electric has presented a task to evaluate the technical skills of potential candidates. The objective of the task is to create a scalable ML inference API. The technical requirements are clearly detailed in the assessment (see Appendix A) and summarized here in Section 4.0. This document serves as a guide to two Git repositories, /nfowler50/pge-assessment-application and /nfowler50/pge-code-pipeline, created as part of a submission for this assessment.

1.2 Table of Contents

1.3 Glossary

Infrastructure as Code (IaC): Automating infrastructure management and provisioning using code and tools instead of manual processes.

Application Programming Interface (API): A set of rules and protocols that allow different software components to communicate with each other, often enabling integration between systems or applications.

AWS Cloud Development Kit (CDK): A framework for defining and provisioning AWS cloud infrastructure using familiar programming languages.

AWS CloudFormation: A service that allows you to define and provision AWS infrastructure using declarative templates.

Elastic Container Registry (ECR): A fully managed Docker container registry that makes it easy to store, manage, and deploy container images in AWS.

Transport Layer Security (TLS): A cryptographic protocol that ensures secure communication and data integrity over a network.

JSON Web Token (JWT): A compact, self-contained token used to securely transmit information between parties as a JSON object.

2.0 Requirements

Category	Requirement	Notes
1. IaC Tooling	-Use Terraform or CloudFormation	Use CloudFormation to deploy IaC written in CDK.
2. Infrastructure	- Provision S3 bucket for model storage. - Deploy EC2 or Lambda for hosting the API. - Use ELB or API Gateway for traffic management. - Configure CloudWatch for monitoring and logging.	Standard serverless stack with s3, Lambda, Api Gateway, and CloudWatch. Secondary stack to meet other requirements with ECS and ELB.
3. Model Hosting	- Deploy pre-trained model using FastAPI or Flask. - Containerize the application with Docker.	Pretrained model stored as Pickle file. Use Flask and Gunicorn to host API. Containerize application with Docker.
4. CI/CD	- Automate Docker image build and push to ECR. - Deploy containers to the hosting environment using CI/CD tools like AWS CodePipeline or GitHub Actions.	CDK to handle all Docker automation and ECR management. AWS CodePipeline to orchestrate CI/CD.
5. Scalability & Optimization	- Implement auto-scaling for EC2 or Lambda. - Use Spot Instances for cost optimization.	Leverage Fargate Spot to simplify autoscaling and server management with benefit of Spot pricing.
6. Security	- Configure IAM roles and policies for least privilege. - Secure API with API keys or token authentication (e.g., JWT).	Use CDK to simplify policy generation for roles and verify least privilege. Create shared JWT secret and login functionality to generate temporary access keys.

2.1. Assumptions

While the assessment in Appendix A is fairly clear on expectations, the simple solution of deploying a serverless stack with Lambda and API Gateway does not require the use of several tools mentioned here, for example Flask, Load Balancing (automatic for Lambda), or Spot instances. To address this, two versions of the application have been developed to showcase a broader skill set: one utilizing API Gateway and Lambda, and the other employing Elastic Load Balancer (ELB) and Elastic Container Service (ECS). Except for shared resources like the pretrained model, these two stacks are designed to remain isolated in the codebase to enhance clarity and maintainability.

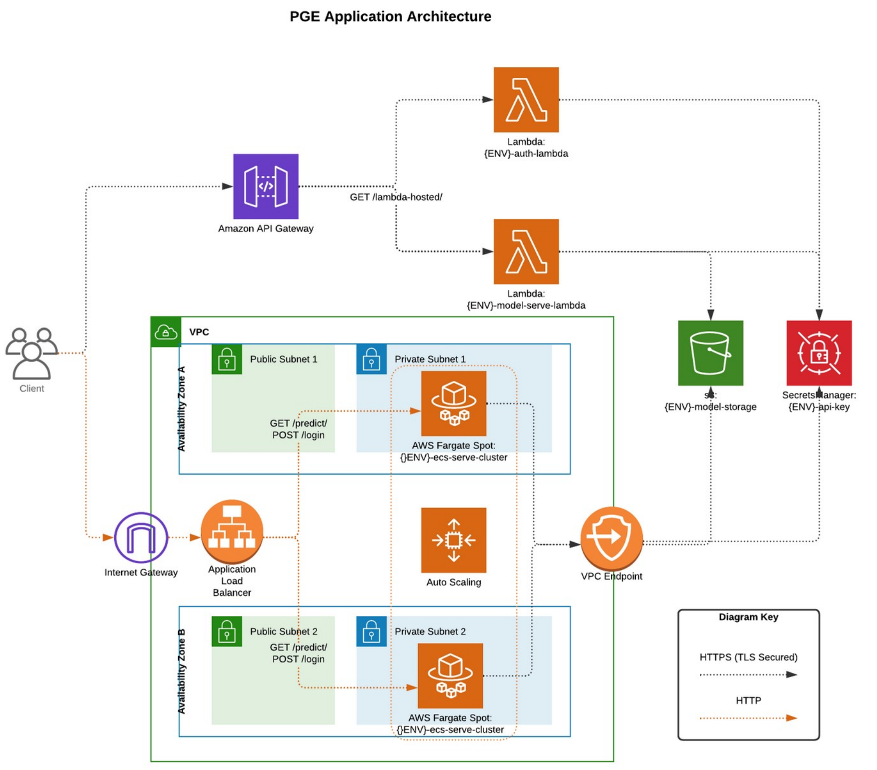
2.2 Out of Scope

Out of Scope	Explanation
Custom Domain	No custom domain is required for the API; the default AWS domain will be used to simplify deployments.

Out of Scope	Explanation
ACM Certificate (TLS)	ACM certificates cannot be published for default ELB domain names (Custom Domain required). Therefore, HTTP communication will be used for the ECS-hosted solution.
User Management System	No user management system has been implemented for this application. For demonstration purposes, login credentials for a single user are hardcoded. However, JWT access tokens are still generated and follow standard expiration practices.
Isolated Production Environment	The CI/CD solution will keep the demo within a single AWS account to simplify reproducibility for the assessor, and avoid complex multi-account setups.
Integration Testing	No integration testing has been implemented. A small series of unit tests have been generated to demonstrate automated testing in deployment pipeline.
Zero Downtime Deployments	Requirements defined general CI/CD, so the added complexity of blue green, canary, or rolling deployments has been avoided. Only basic CI/CD has been implemented here, meaning that pushing or rolling back changes will result in some service downtime.
Automated Rollback	Requirements defined general CI/CD, so the added complexity of automating rollback by monitoring alarms has been avoided. Any failures in production service would remain active until rollback to previous version had been initiated.

3.0 Solutions

3.1 Application Architecture



Two interfaces have been implemented: one using AWS API Gateway for a Lambda-hosted solution, and another using an Application Load Balancer for an ECS-hosted solution.

For both approaches, two routes are defined: `/login` and `/predict`.

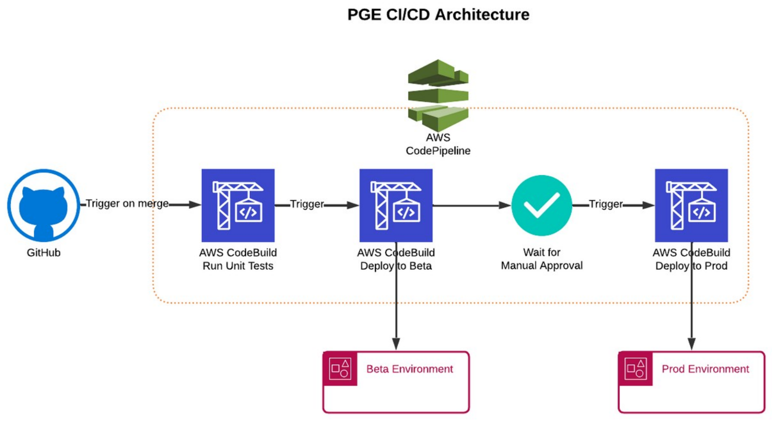
- **/login**: This route accepts POST requests, which are routed to a compute resource. The service validates the provided username and password, then generates and returns a temporary JWT access token.
- **/predict**: This route accepts GET requests, which are routed to a compute resource responsible for generating predictions. The service validates the JWT access token, processes the input data, and returns a prediction based on the input.

The compute resources for login have access to a JWT secret key stored in AWS Secrets Manager, which is used to sign and generate the temporary JWT access token. The compute resources for prediction also have access to the same JWT secret key for token validation, and they retrieve a pre-trained model stored as a Pickle file in S3 to generate predictions.

The Lambda-hosted solution is fronted by an AWS API Gateway, with AWS Lambda serving as the compute resource. All traffic is secured using TLS.

The ECS-hosted solution is fronted by an Application Load Balancer, with ECS Fargate Spot instances as the compute resource. Communication with the Application Load Balancer is not protected by TLS and is transmitted over unprotected HTTP (see Section 5 for Out of Scope details and section 8 for Security Risks). The ECS Fargate instances are managed in an autoscaling group using Spot instances, distributed across two availability zones to enhance reliability. The Application Load Balancer is hosted in a public network, accepting traffic from the internet, while the ECS Fargate instances are hosted in a private network, allowing traffic only from the Application Load Balancer.

3.2 CI/CD Architecture



A continuous integration and continuous deployment (CI/CD) pipeline has been implemented with GitHub and AWS CodePipeline. The pipeline has five stages; **Source, Testing, Deploy to Beta, Wait for Approval,** and **Deploy to Prod**.

- **Source:** Detects changes in source repository and triggers the orchestration pipeline in AWS.

- **Testing:** Executes unit tests on scripts in the repository using a container managed by AWS CodeBuild. Test results determine the pipeline's next steps, and detailed logs are available in the AWS console for analysis and debugging.
- **Deploy to Beta:** Synthesizes a CloudFormation template from the CDK script within a container managed by AWS CodeBuild. The template is then deployed via CloudFormation to Beta environment. Deployment success or failure dictates subsequent pipeline actions.
- **Wait for Approval:** Includes a manual approval step as a precaution. The pipeline pauses here until approval is granted.
- **Deploy to Prod:** Similar to the "Deploy to Beta" stage, this step synthesizes and deploys a CloudFormation template from the CDK script through AWS CodeBuild and CloudFormation.

Numerous enhancements could improve this pipeline, such as automating ML model training and hyperparameter tuning, implementing zero-downtime deployments with blue-green, canary, or rolling strategies, or automating rollbacks if key performance metrics fail to meet specified thresholds.

3.3 Security Considerations

Security Consideration	Current Implementation
IAM Policies	Principal of least privilege has been implemented, with the exception of the deployment role which has been left over privileged to allow an assessor to more easily deploy this application in their own account.
Resource Policies	While public access has been blocked, access for entities within the account could be more heavily restricted with specified resource policies.
Networking	Virtual Private Cloud has been implemented where it is relevant. Infrastructure is layered with containers running in private subnets and traffic routed through load balancers hosted in public subnets.
API Authentication	While credentials and the API secret key are hardcoded for demonstration purposes, JWT access tokens are properly generated with expiration. These stateless tokens are created during login events and grant access to other resources upon validation.

3.4 Risks and Open Issues

Security Risk	Mitigation in Production Environment
HTTP (non-TLS)	TLS has not been implemented for load balancer because of certificate restrictions. In production, we would generate an ACM certificate for relevant domain, assign to load balancer, restrict traffic to port 443, and implement a redirect for any port 80 traffic.
Hardcoded JWT Secret Key	JWT secret is properly stored in AWS Secrets Manager, but it is hardcoded in the infrastructure. This should be generated outside of the deployment and not reused for all environments.
Unrestricted VPC NACL	Assuming relevant clients are known sources, implement VPC Network ACLs with least privilege rules, restricting inbound and outbound traffic to

Security Risk	Mitigation in Production Environment
	only necessary IPs.
Under restricted Resource Policies	S3 bucket has been generated with “Block All Public Access” to True, but that does not restrict inter account access. Further protection could be provided by updating a restrictive resource policy.

3.5 Scalability and Optimization

Optimization is a major topic in system design, but it always comes down to context and trade offs. Given that the context here is a demo app to meet requirements defined in an assessment, the optimal solution will be defined as follows:

Minimize cost and level of effort of hosting the demo application while exposing configurations that could allow massive scalability if needed.

Regarding the two architectures, the primary configurations impacting scalability are related to the compute resources. Both the API Gateway and Elastic Load Balancer are highly scalable by default.

The model being served through AWS Lambda can technically scale as is, but further configurations could be made to increase default concurrency limitations if some threshold was broken. To address this, a metric and alarm have been generated to monitor Lambda concurrency and alert if a threshold of 80% max concurrency is reached.

Several contributions have been made to optimize the Lambda’s performance and minimize latency. First, model initialization has been moved outside the handler. This means that only on warm up (when a new container is starting) will the s3 model be imported and loaded. Once initialized, only the handler will be invoked on sequential calls. Further, to eliminate delays/timeouts from cold starts, an AWS Events timer has been created to ping the Lambda function every 5 minutes. There are better ways to do this in production, like setting minimum concurrency, but because this demo app will see very little traffic, using an Events timer like this will keep costs at a minimum without causing service timeouts for assessors.

The model being served through AWS Elastic Container Service has been implemented within an auto scaling group, but the current configuration limits autoscaling to two tasks. Further configurations could be made to increase autoscaling limit with minimal effort. To address any issues with autoscaling limit, a metric and alarm has been created to alert if max capacity is reached.

5.0 Monitoring

Basic monitoring resources have been generated for this application. The resources include error and latency metrics, alarms, and dashboards.

Metrics generated include:

- Lambda error tracking for authentication and prediction methods.
- 4xx error tracking for authentication failures.
- 5xx error tracking for service failures.
- Latency tracking for application response latency.
- (ECS only) CPU utilization
- (ECS only) Memory utilization

Alarms generated track metrics against thresholds over 60 second periods. Alarms generated include:

- Lambda error alarms, to alert if errors break threshold of 5 errors/min.
- 4xx alarm, to alert if 4xx errors break threshold of 10 errors/min.
- 5xx alarm, to alert if 5xx errors break threshold of 5 errors/min.
- Latency alarms, if latency exceeds 2 seconds.
- (ECS only) Memory utilization alarm if exceeds 70%.
- (ECS only) CPU utilization alarm if exceeds 80%.

These metrics have all been surfaced in dashboards, one for ECS hosted application and one Lambda hosted application. These dashboards can be found in the AWS CloudWatch console by selecting “Dashboards” in the left hand column.

Appendix A: Assessment Responses

Scenario: Deploying a Scalable ML Inference API
Objective: Build a scalable API on AWS to serve a pre-trained machine learning model.

Key Requirements:
1. Infrastructure Setup (IaC)
Use Terraform or CloudFormation to provision:
    - S3 bucket for storing the ML model.
    - EC2 instance or AWS Lambda to host the API.
    - Elastic Load Balancer or API Gateway for traffic management.
    - CloudWatch for monitoring and logging.

2. Model Hosting
    - Deploy a pre-trained ML model using FastAPI or Flask.
    - Containerize the application with Docker.

3. CI/CD Pipeline
Automate deployment using Jenkins, AWS CodePipeline, GitHub Actions, or equivalent:
    -Build and push Docker images to ECR.
    -Deploy updated containers to the hosting environment.

4. Scalability and Optimization
    -Implement auto-scaling and leverage Spot Instances for cost efficiency.

5. Security
    - Configure IAM roles/policies.
    - Secure API access with API keys or token authentication.

Deliverables:
1. Code Repository: Include IaC, application code, and pipeline configurations.
2. Documentation: A README with architecture details, deployment steps, and API testing instructions.
3. Optional: A URL to the running API.

Assessment Focus:

-Completeness of the solution.
-Code quality and structure.
-Security practices and cost-efficiency.
-Scalability and ease of deployment.

Appendix B: Navigating the application repository

- */app.py*: this script defines and orchestrates high level stack to be deployed.

- */pge_app_infrastructure*: this directory contains all substack IaC definitions.
- */pge_app_infrastructure/pge_stack.py*: defines shared resources for entire stack.
- */pge_app_infrastructure/ecs_hosted.py*: defines application hosted in ECS through Application Load Balancer.
- */pge_app_infrastructure/ecs_monitoring_stack.py*: defines metrics, alarms, and dashboards for ECS hosted application.
- */pge_app_infrastructure/lambda_hosted.py:* defines application hosted in Lambda through API Gateway.
- /pge_app_infrastructure/lambda_monitoring_stack.py: defines metrics, alarms, and dashboards for lambda hosted application.

- */ecs-hosted*: Dockerfile and Python scripts for ECS container supporting login and predict methods.
- */lambda-auth*: Dockerfile and Python scripts for Lambda used for login method.
- */lambda-hosted*: Dockerfile and Python scripts for Lambda used for predict method.
- */model*: Python scripts and data used to generate pretrained model

Appendix C: Navigating the pipeline repository

- */app.py*: this script defines and orchestrates high level stack to be deployed.
- */code_pipeline*: this directory contains all substack IaC definitions.
- */codepipeline/code_pipeline_stack.py*: defines CodePipeline infrastructure and stages.

Appendix D: Instructions to install and configure AWS CLI & CDK

1. Install and Configure AWS CLI *(not required if already installed/configured)*

If you do not already have the AWS CLI installed , please follow the instructions shared here: https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

To configure the AWS CLI, you will need to run the following command at the command line:

```
aws configure
```

You will be prompted for:

- AWS Access Key ID

  - AWS Secret Access Key
  - Default region name (e.g., `us-east-1`)
  - Default output format (e.g., `json`)

For detailed instructions, refer to the AWS CLI Installation and Configuration Guide.

2. Install AWS CDK

- **Install AWS CDK** globally using npm:

  ```
  npm install -g aws-cdk
  ```

If you don't have `npm` installed, you can install Node.js and npm from Node.js website.

- **Verify CDK installation**: After installing, verify that AWS CDK is installed successfully by running:

  ```
  cdk –version
  ```

Appendix E: Instructions to Deploy directly (no pipeline)

To deploy this application directly, without the use of the AWS CodePipeline:

1. Clone the Git Repository

First, clone the Git repository containing the AWS CDK stack:

```
git clone https://github.com/nfowler50/pge-assessment-application
cd pge-assessment-application
```

2. Set Up Python Virtual Environment

- **Create and activate a virtual environment**: In your project directory, create and activate the virtual environment:

  ```
  python3 -m venv .venv
  source .venv/bin/activate  # On macOS/Linux
  .venv\Scripts\activate     # On Windows
  ```

- **Install dependencies**: Once the virtual environment is activated, install the dependencies listed in `requirements.txt`:

  ```
  pip install -r requirements.txt
  ```

  This ensures that the necessary Python packages for AWS CDK are installed.

3. Deploy the AWS CDK Stack

- **Bootstrap your environment** (if not done previously): If this is your first time deploying with CDK, you need to bootstrap your AWS environment to set up resources for CDK to use:

  ```
  cdk bootstrap
  ```

- **Deploy the CDK stack**: After bootstrapping (if necessary), you can deploy your stack:

  ```
  cdk deploy
  ```

  This command will deploy the resources specified in your CDK app to your AWS account.

Note: you will be prompted to confirm the creation of several roles and policies.

4. Monitor the Deployment

The `cdk deploy` command will output the progress of your deployment. You can also check the AWS CloudFormation Console for details about the stack and the resources being created.

By following these steps, you'll be able to clone the repository, configure AWS CLI, set up a Python virtual environment, install dependencies, and deploy the AWS CDK stack.

Appendix F: Instructions to deploy active pipleline

Deploying the CI/CD pipeline stack is very similar to deploying the main stack directly. Once this pipeline is deployed, it will automatically be triggered when any changes are merged to the main branch of repository.

1. Clone the Git Repository

First, clone the Git repository containing the AWS CDK stack:

```
git clone https://github.com/nfowler50/pge-code-pipeline
cd pge-code-pipeline
```

2. Set Up Python Virtual Environment

- **Create and activate a virtual environment**: In your project directory, create and activate the virtual environment:

  ```
  python3 -m venv .venv
  source .venv/bin/activate  # On macOS/Linux
  .venv\Scripts\activate     # On Windows
  ```

- **Install dependencies**: Once the virtual environment is activated, install the dependencies listed in `requirements.txt`:

  ```
  pip install -r requirements.txt
  ```

  This ensures that the necessary Python packages for AWS CDK are installed.

3. Deploy the AWS CDK Stack

- **Bootstrap your environment** (if not done previously): If this is your first time deploying with CDK, you need to bootstrap your AWS environment to set up resources for CDK to use:

  ```
  cdk bootstrap
  ```

- **Deploy the CDK stack**: After bootstrapping (if necessary), you can deploy your stack:

  ```
  cdk deploy
  ```

  This command will deploy the resources specified in your CDK app to your AWS account.

Note: you will be prompted to confirm the creation of several roles and policies.

4. Monitor the Deployment

The `cdk deploy` command will output the progress of your deployment. You can also check the AWS CloudFormation Console for details about the stack and the resources being created.

Once this pipeline is deployed, it will be automatically triggered when any changes are merged to the main branch of application repository.

Appendix G: Deleting the Stack

To delete any stack that you have deployed directly, whether its the application stack or pipeline stack, simply navigate to the root directory (/pge-assessment-app/ or /pge-codepipeline) and type the following command:

```
cdk destroy --all
```

If a stack has been deployed via the pipeline, and not directly, you will need to navigate to the CloudFormation console in AWS, select the stack to delete, then select “Delete”.

Appendix H: Instructions to send request / test endpoints

No deployment is necessary to test the demo endpoints. A script has been written to allow you to send requests to endpoints with minimal effort. This script can be found in the application repository at:

/pge-assessment-application/hit-live-endpoints.py

1. Run the hit-live-endpoint script provided

Navigate to the /pge-assessment-application/ directory in your cloned repository (see Appendix E step 1 on how to clone) and run the following command:

python hit-live-endpoints.py

2. Input base URL of endpoint you would like to test

You will be prompted to provide a base URL. Demo endpoints provided by Nick Fowler’s deployment are listed below, simply copy and paste one of the following as an input:

API Gateway (lambda hosted) base URL:
ALB (ECS-hosted) base URL:

(Optional alternative) If you'd like to send requests to your own deployment, the base URL of the API Gateway and Application Load Balancer is available through the CloudFormation console. Navigate to the console and select one of the following:
{SANDBOX/BETA/PROD}-ECS-hosted-stack
{SANDBOX/BETA/PROD}-Lambda-hosted-stack
Your endpoint URL will be located under the "Outputs" tab

3. Receive initial response
On input of base URL, you should receive a response that says "Login Successful" and prints out the returned Access Key. This access key is a temporary JWT token that can be used for successive requests.

4. Provide some value between 0 and 4 (inclusive)
Once authenticated, you should be prompted to provide some input between 0 and 4. A prediction from pretrained model should be returned.

That's it! Technically the JWT access key is still valid, but as an FYI, if you run the hit-live-endpoints script more than once, each time it will log in again (just a basic demo script).