Federico Rodríguez
1000752822

Practical Component (50%)
Code repo: https://github.com/Corosso/tech_test_Cadena

1. Python Exercise:

For this exercise, I implemented an adaptive solution based on input characteristics, because we could use different algorithms to minimize time execution and memory management.
First, is implemented a linear search algorithm, this is a simple algorithm using a while loop to iterate through the array.
This is useful for small datasets (less than 10,000 elements) and numbers with less than 5 digits (99,999 or less)

Second, is implemented an algorithm called "Sieve of Eratosthenes", which is an efficient method for finding all prime numbers up to a given limit.
This algorithm uses a boolean array to mark non-prime numbers and eliminates multiples of each prime iteratively.
It is more suitable for large datasets (10,000 elements or more) and when working with numbers greater than 5 digits, as it significantly reduces time complexity compared to a linear search approach.

Third, not implemented in the exercise but is part of my solution, the Segmented Sieve algorithm would be an ideal choice for generating prime numbers within very large ranges, where storing all values in memory is not viable.
This approach divides the range into smaller segments and applies the Sieve of Eratosthenes on each part independently, optimizing both memory consumption and performance.
It is particularly useful when working with extremely large datasets (100,000 entries) where efficiency and scalability become critical.

2. C# Exercise:

For this exercise, I choose the Two-Pointer algorithm as the main approach to check whether a string is a palindrome.
This algorithm operates with **O(n)** time complexity and **O(1)** space complexity, as it uses two pointers that move toward the center of the string while comparing characters directly.
It is efficient for all string sizes, avoids unnecessary memory allocation, and terminates early upon detecting a mismatch.

AWS Services and IT Development Concepts from Developer's Perspective (50%)

1. Firstable, Amazon RDS is a relational database service, while Amazon DynamDB is for no relational (NoSQL) databases.
Also RDS supports traditional databases such as MySQL, Postgres, etc. Otherwise, DynamoDB uses a schema where data is stored as key-value pairs, similar to JSON documents. The way to interact with these databases is through an API.

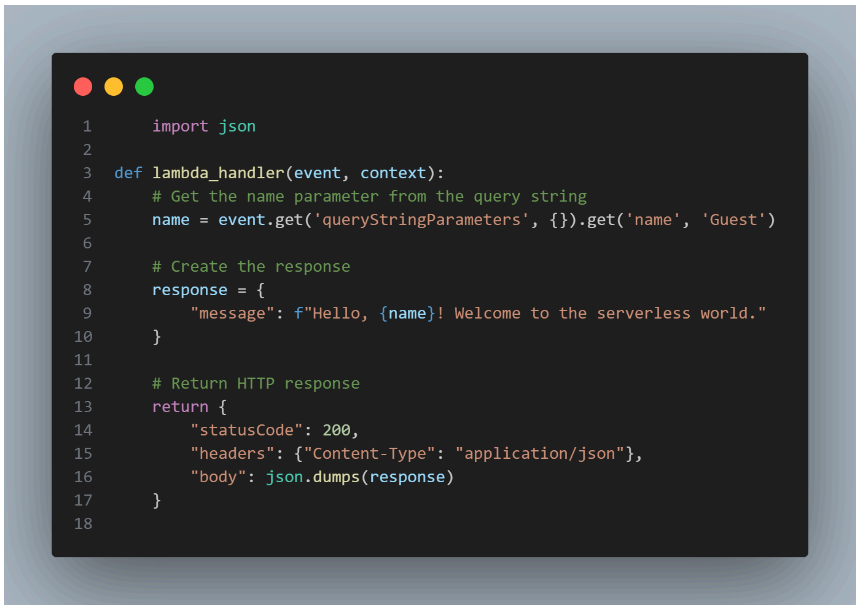
RDS is certainly more appropriate when solutions have more complexity, this complexity is reflected on relationships, data consistency, strict data and complex queries. Some examples of these cases could be e-commerce systems or content management applications.

Dynamo instead, is used on systems that demand low-latency and flexible data models, for most cases is used on online leaderboards and IoT telemetry.

2. AWS Lambda is a computer service that allows developers to run code without any server, this means we can forget about managing this. This brings us to the second question, on what's called serverless computing, as its name implies, we just need to create and upload our code. Lambda is in charge of these "complex tasks", such as server setup, scaling, fault tolerance and maintenance. Also, Lambda takes care to respond to events, such as API calls, database updates, etc.

For this example, we have a simple REST API that returns a greeting message.

The API Gateway receives the request and triggers the AWS Lambda function.



```python
    import json

def lambda_handler(event, context):
    # Get the name parameter from the query string
    name = event.get('queryStringParameters', {}).get('name', 'Guest')

    # Create the response
    response = {
        "message": f"Hello, {name}! Welcome to the serverless world."
    }

    # Return HTTP response
    return {
        "statusCode": 200,
        "headers": {"Content-Type": "application/json"},
        "body": json.dumps(response)
    }

```

3. Well, DevOps is a relatively new concept in software development, this aims to combine development and operations (as its name is), to reduce the gap between them.
Basically what DevOps do is increase collaboration, reduce delivery times and improve quality and reliability on software solutions.

The main problem before having DevOps was that after an application finished its development process from the developers, it is handed to operations teams to deploy and maintain, this leads to delays, inconsistency or whatever other problem you can think about when 2 different teams develop and maintain an application with barely any contact.

AWS provides a complete stack of services and tools that aims for DevOps practices.

AWS CodeCommit, is a git based repository allowing developers to use version control and centralize code, basically GitHub but from and for AWS ecosystem.

AWS CodeBuild, is a build service that allows code compiling, running tests and producing deployment-ready artifacts.

AWS CodeDeploy, as its name says, this tool automates the process of deploying code, the main advantage of this is that you can perform roll backs if anything goes wrong and goes back to when it was working.

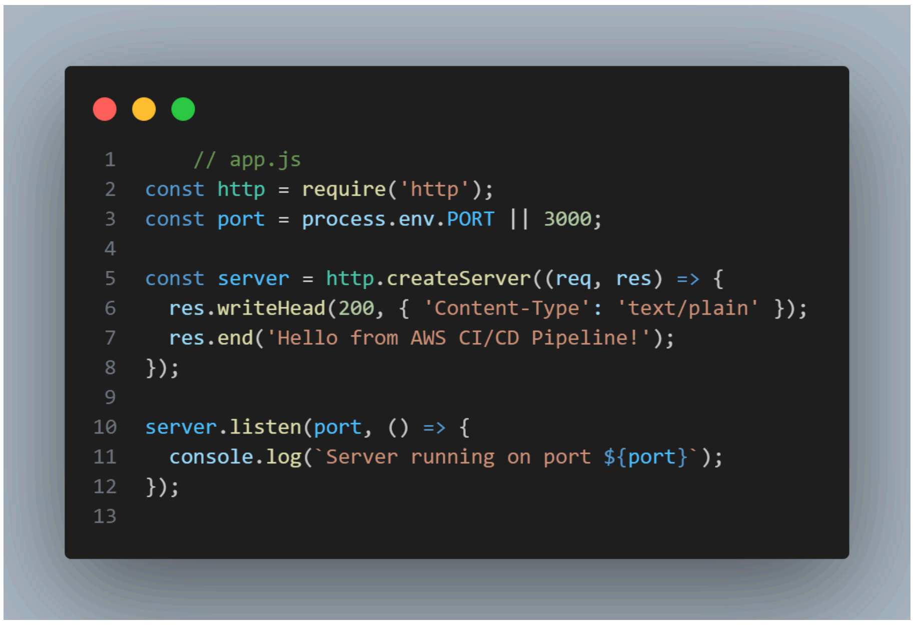
AWS CodePipeline, is basically a tool that connects all stages, source control, build, test and deployment, it combines all of the tools above to create a complete CI/CD workflow.

4. A CI/CD process is basic.

1. The source stage, here the pipeline in triggered when new code is pushed from the source control
2. The build stage is when AWS CodeBuild (for this AWS scenario) compiles the code, installs necessary dependencies, and runs tests. Also you can add additional calls, functions, etc.
3. Deploy stage, The build artifact is deployed to an environment

This is the example of how to deploy a web application

First this is our app

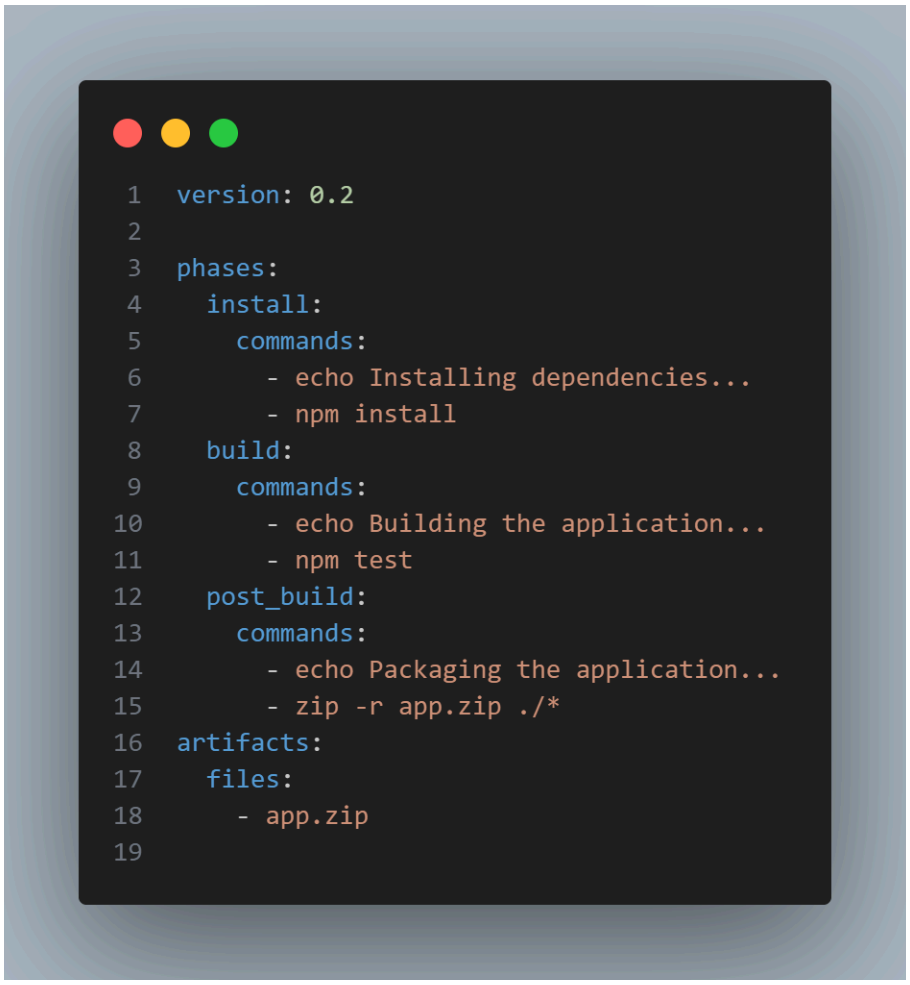


```javascript
    // app.js
const http = require('http');
const port = process.env.PORT || 3000;

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from AWS CI/CD Pipeline!');
});

server.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

Then the YAML (buildspec.yml) where are the instructions to install dependencies, runs test a create a .zip artifact that will be deployed

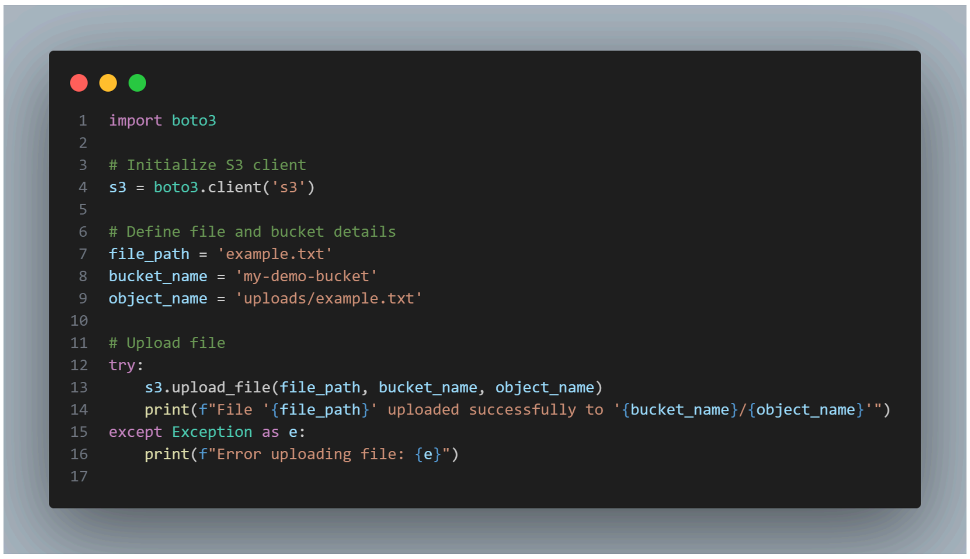


```yaml
version: 0.2

phases:
  install:
    commands:
      - echo Installing dependencies...
      - npm install
  build:
    commands:
      - echo Building the application...
      - npm test
  post_build:
    commands:
      - echo Packaging the application...
      - zip -r app.zip ./*
artifacts:
  files:
    - app.zip
```

5. Amazon S3, S3 for Simple Storage Service, as its name says, is a storage service, whether you can store and retrieve any amount of data, this data is stored as objects inside buckets which are containers for these objects (objects can be data, keys or metadata).
S3 is designed for durability, availability and reliability.
The most common use cases are static website hosting, backups, data lakes or content distributors

Here's an example for uploading a file to S3



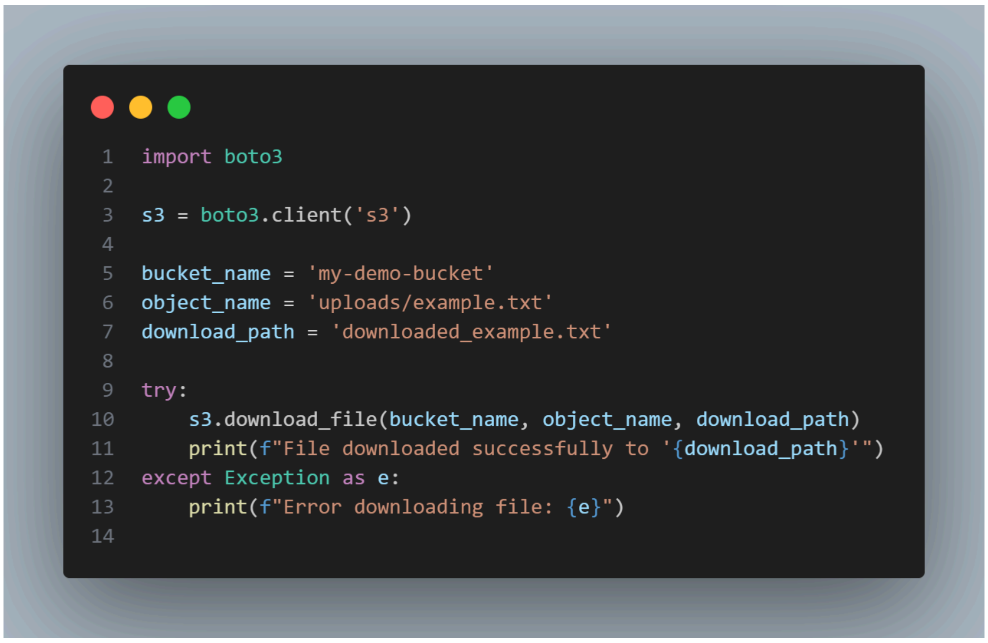
```python
import boto3

# Initialize S3 client
s3 = boto3.client('s3')

# Define file and bucket details
file_path = 'example.txt'
bucket_name = 'my-demo-bucket'
object_name = 'uploads/example.txt'

# Upload file
try:
    s3.upload_file(file_path, bucket_name, object_name)
    print(f"File '{file_path}' uploaded successfully to '{bucket_name}/{object_name}'")
except Exception as e:
    print(f"Error uploading file: {e}")
```

Here's an example for downloading a file from S3



```python
import boto3

s3 = boto3.client('s3')

bucket_name = 'my-demo-bucket'
object_name = 'uploads/example.txt'
download_path = 'downloaded_example.txt'

try:
    s3.download_file(bucket_name, object_name, download_path)
    print(f"File downloaded successfully to '{download_path}'")
except Exception as e:
    print(f"Error downloading file: {e}")
```