



UNIVERSIDAD INTERAMERICANA DE PANAMÁ

Proyecto Final: Bot de Orientación Vocacional "Miguel"

Grupo:

Hojaldre Team

Integrantes del proyecto:

Abraham Btesh

Alexa Cuevas

Eynar Gonzalez

Ariel Torralgo

Docente:

Abdel Rivera

Materia:

Programación V

Año:

2025

Tabla de Contenidos

| | |
|--|----|
| Introducción | 3 |
| Objetivos..... | 3 |
| Descripción General | 4 |
| Infraestructura AWS | 6 |
| Estructura del Código | 8 |
| Integración con Telegram..... | 9 |
| Integración con OpenAI | 10 |
| Manejo de Datos en DynamoDB | 12 |
| Flujo Completo de Ejecución | 14 |
| Seguridad | 17 |
| Proceso de Despliegue | 18 |
| Monitoreo y Mantenimiento | 19 |
| Mejoras Futuras | 20 |
| Apéndices | 21 |
| Referencia de APIs | 28 |

Introducción

Este documento presenta la documentación técnica del proyecto "Miguel", un bot de Telegram para orientación vocacional desarrollado como proyecto final para la asignatura de Programación V de la Universidad Interamericana de Panamá.

El proyecto surge como respuesta a la necesidad de los estudiantes de recibir orientación personalizada al elegir una carrera universitaria. Miguel utiliza inteligencia artificial para analizar los intereses y aptitudes de los usuarios, recomendando específicamente carreras disponibles en la Universidad Interamericana de Panamá (UIP).

Objetivos

- Desarrollar un asistente virtual accesible vía Telegram que recomiende carreras de la UIP según el perfil del usuario
- Implementar un sistema de conversación contextual para identificar intereses vocacionales
- Utilizar inteligencia artificial para analizar respuestas y generar recomendaciones precisas
- Crear una arquitectura serverless en AWS que garantice disponibilidad y eficiencia

El sistema incluye un bot funcional en Telegram, integración con OpenAI, persistencia de datos para mantener el contexto de las conversaciones y despliegue en infraestructura AWS.

Esta documentación detalla los componentes técnicos, arquitectura, código y procesos de implementación del sistema.

Descripción General

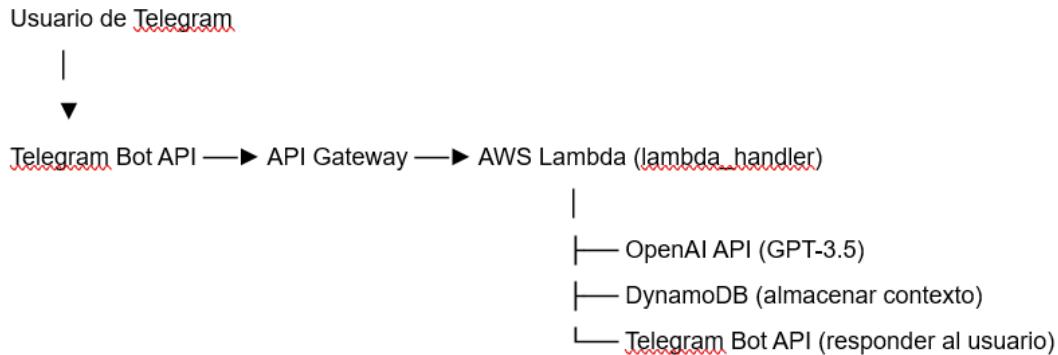
Esta Lambda implementa un bot de asesoría vocacional personalizado llamado "Miguel", que responde a mensajes enviados por usuarios en Telegram. Utiliza OpenAI (ChatGPT) para generar respuestas personalizadas con base en un conjunto de reglas estrictas, orientando a los estudiantes sobre cuál carrera universitaria podrían estudiar en la Universidad Interamericana de Panamá (UIP). La conversación se guarda en DynamoDB para mantener el contexto.

El bot está diseñado para:

- Realizar preguntas específicas sobre intereses y aptitudes de los estudiantes
- Analizar las respuestas para identificar carreras adecuadas
- Recomendar entre 1 y 3 carreras oficiales de la UIP
- Explicar por qué esas carreras serían adecuadas para el estudiante
- Mantener una conversación contextual y natural

Arquitectura del Sistema

El sistema está compuesto por los siguientes componentes principales interconectados:



Flujo de datos:

1. El usuario envía un mensaje al bot de Telegram
2. Telegram reenvía el mensaje al webhook configurado (API Gateway)
3. API Gateway pasa la solicitud a la función Lambda
4. Lambda procesa el mensaje:
 - Extrae el ID del chat y el texto del mensaje
 - Recupera el historial de conversación de DynamoDB
 - Prepara la solicitud para OpenAI
 - Envía la solicitud a OpenAI API
 - Recibe la respuesta generada
 - Actualiza el historial en DynamoDB
 - Envía la respuesta al usuario a través de la API de Telegram

Infraestructura AWS

AWS Lambda

Configuración general:

- **Runtime:** Python (versión específica no especificada, probablemente 3.8+)
- **Memoria:** 256 MB
- **Almacenamiento efímero:** 512 MB
- **Tiempo de espera:** 45 segundos
- **SnapStart:** None

Variables de Entorno:

| Variable | Descripción |
|---------------------|---|
| DYNAMODB_TABLE_NAME | Nombre de la tabla DynamoDB que almacena sesiones |
| OPENAI_API_KEY | API Key de OpenAI |
| TELEGRAM_BOT_TOKEN | Token del bot de Telegram |
| ASSISTANT_ID | ID del asistente (si se usa API de asistentes) |

API Gateway

- **Tipo:** REST API
- **Configuración del Endpoint:** Recibe webhooks de Telegram en un endpoint dedicado (ej: /webhook)
- **Método HTTP:** POST
- **Integración:** Lambda Proxy Integration
- **Seguridad:** Sin autenticación adicional (se basa en la seguridad del webhook de Telegram)

Amazon DynamoDB

- **Nombre de la tabla:** Especificado en la variable de entorno DYNAMODB_TABLE_NAME
- **Clave primaria:** chat_id (String) - ID único del chat de Telegram
- **Modo de capacidad:** Bajo demanda (on-demand)
- **Índices secundarios:** Ninguno
- **TTL:** No configurado (potencial mejora futura)

Estructura del Código

Estructura del Proyecto

```
lambda_function/
|—— lambda_function.py # Función principal y lógica
└—— Layer.zip          # Librerías externas
```

Dependencias (Lambda Layer)

Se creó un Lambda Layer personalizado con las siguientes bibliotecas:

bash

```
# Comando para crear el Layer
mkdir -p ~/compatible_layer/python
pip install --target ~/compatible_layer/python openai==0.28.1 boto3 urllib3 requests
cd ~/compatible_layer
zip -r ..../compatible_layer.zip python/
```

Librerías incluidas:

- openai==0.28.1 – Cliente para interactuar con ChatGPT
- boto3 – Cliente para acceder a DynamoDB desde Lambda
- urllib3, requests – Utilizadas para llamadas HTTP (como a Telegram)

Integración con Telegram

Configuración del Bot

1. Crear un bot con @BotFather en Telegram
2. Obtener el token del bot
3. Configurar el webhook para recibir mensajes:

https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/setWebhook?url={API_GATEWAY_URL}

Formato de mensaje entrante (Webhook)

```
json
{
  "body": "{\"message\": {\"chat\": {\"id\": 123456}, \"text\": \"Hola\"}}"
}
```

Envío de respuestas

Se utiliza la función send_message_telegram(chat_id, text) que hace una solicitud HTTP POST al endpoint de Telegram:

https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage

Con el siguiente payload:

```
json
{
  "chat_id": "123456",
  "text": "Texto de respuesta del asistente"
}
```

Integración con OpenAI

El sistema utiliza la API de ChatCompletion de OpenAI (versión 0.28.1) para generar respuestas contextuales.

Formato de solicitud a OpenAI

```
python
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": SYSTEM_INSTRUCTIONS},
        {"role": "user", "content": "mensaje del usuario"},
        {"role": "assistant", "content": "respuesta anterior del asistente"},
        # ... más mensajes del historial
        {"role": "user", "content": "mensaje actual del usuario"}
    ],
    temperature=0.7,
    max_tokens=600
)
```

Instrucciones del Sistema (System Prompt)

Las instrucciones del sistema (SYSTEM_INSTRUCTIONS) definen el comportamiento del asistente Miguel:

- **Misión Principal:** Ayudar a descubrir carreras universitarias que se ajusten al perfil del usuario

- **Reglas Estrictas:**
 - No inventar carreras fuera de la lista proporcionada
 - Solo recomendar carreras de la UIP
 - Hacer preguntas específicas para entender al usuario
 - Usar lenguaje claro y amigable
- **Formato de Respuesta:** Estructura definida para las recomendaciones
- **Lista de Carreras:** Organizadas por categorías:
 - Ciencias de la Salud
 - Ciencias Administrativas, Marítima y Portuaria
 - Ingeniería, Arquitectura y Diseño
 - Hotelería, Gastronomía y Turismo
 - Derecho y Ciencias Políticas

Manejo de Datos en DynamoDB

Estructura de datos

Tabla: Nombre definido en variable de entorno DYNAMODB_TABLE_NAME **Clave primaria:** chat_id (String)

Formato de item:

json

```
{  
    "chat_id": "123456",  
    "conversation": [  
        {"role": "user", "content": "Hola"},  
        {"role": "assistant", "content": "Hola, soy Miguel..."},  
        {"role": "user", "content": "Me gustan las matemáticas"},  
        {"role": "assistant", "content": "Gracias por compartir..."}  
    ]  
}
```

Operaciones de lectura/escritura

Lectura:

python

```
response = table.get_item(Key={'chat_id': chat_id})  
  
if 'Item' in response and 'conversation' in response['Item']:  
    return response['Item']['conversation']  
  
return []
```

Escritura:

```
python  
table.put_item(Item={  
    'chat_id': chat_id,  
    'conversation': conversation  
})
```

Gestión del contexto

- Se mantienen los últimos 10 mensajes para evitar exceder límites de tokens de OpenAI
- El formato es compatible con la estructura esperada por la API de OpenAI
- Se actualiza después de cada intercambio de mensajes

Flujo Completo de Ejecución

1. Recepción de mensaje (`lambda_handler`)

```
python
def lambda_handler(event, context):
    try:
        # Extraer datos del mensaje
        body = json.loads(event["body"])
        message = body['message']
        chat_id = str(message['chat']['id'])
        user_message = message['text']

        # Procesar mensaje y generar respuesta...
        return {
            'statusCode': 200,
            'body': json.dumps('Message processed successfully!')
        }
    except Exception as e:
        print(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps({'error': str(e)})
        }
```

2. Recuperación del historial

```
python  
conversation_history = get_conversation_history(chat_id)
```

3. Procesamiento y generación de respuesta

```
python  
# Añadir mensaje del usuario al historial  
conversation_history.append({"role": "user", "content": user_message})  
  
# Limitar el historial a los últimos 10 mensajes para controlar tokens  
if len(conversation_history) > 10:  
    conversation_history = conversation_history[-10:]  
  
# Preparar los mensajes para la API  
messages = [  
    {"role": "system", "content": SYSTEM_INSTRUCTIONS}  
] + conversation_history  
  
# Obtener la respuesta de ChatGPT  
response = openai.ChatCompletion.create(  
    model="gpt-3.5-turbo",  
    messages=messages,  
    temperature=0.7,  
    max_tokens=600  
)
```

```
# Extraer la respuesta
assistant_response = response.choices[0].message.content
```

4. Actualización del historial y respuesta

```
python
```

```
# Añadir la respuesta al historial
```

```
conversation_history.append({"role": "assistant", "content": assistant_response})
```

```
# Guardar el historial actualizado en DynamoDB
```

```
save_conversation_history(chat_id, conversation_history)
```

```
# Enviar la respuesta al usuario en Telegram
```

```
send_message_telegram(chat_id, assistant_response)
```

Seguridad

Consideraciones de seguridad implementadas:

- Uso de variables de entorno para almacenar claves y tokens sensibles
- Timeout adecuado para evitar costos excesivos o ataques de denegación de servicio
- Manejo apropiado de errores para evitar filtraciones de información

Recomendaciones adicionales:

- No loguear la API Key ni los tokens
- Agregar protección a la URL de webhook (mediante IP whitelist, verificación de Telegram, etc.)
- Control de errores robusto para evitar fallos silenciosos
- Utilizar AWS Secrets Manager en lugar de variables de entorno para mayor seguridad
- Implementar limitación de tasa en API Gateway

Proceso de Despliegue

Actualmente, el despliegue es manual a través de la consola de AWS:

1. DynamoDB:

- Crear tabla con chat_id como clave primaria
- Configurar modo de capacidad bajo demanda

2. Lambda:

- Crear función Lambda con el runtime adecuado
- Configurar memoria (256 MB), timeout (45 segundos) y almacenamiento efímero (512 MB)
- Subir el código de lambda_function.py
- Configurar el Lambda Layer con las dependencias
- Configurar variables de entorno

3. API Gateway:

- Crear una API REST
- Configurar recurso y método POST
- Integrar con la función Lambda
- Desplegar la API
- Obtener la URL del endpoint

4. Telegram:

- Configurar el webhook con la URL del API Gateway

Monitoreo y Mantenimiento

Estado actual:

Actualmente no hay monitoreo formal implementado.

Recomendaciones para monitoreo:

- Configurar logs de CloudWatch para seguimiento detallado
- Crear métricas personalizadas para medir uso y desempeño
- Implementar alertas para errores frecuentes o fallas en la integración
- Monitorear costos de OpenAI API para evitar cargos inesperados

Mantenimiento:

- Actualizar periódicamente las dependencias, especialmente para correcciones de seguridad
- Revisar y optimizar las instrucciones del sistema conforme evolucionen las necesidades
- Verificar y actualizar la lista de carreras si hay cambios en la oferta educativa de la UIP

Mejoras Futuras

1. Funcionalidad:

- Agregar validación de idioma y comandos especiales
- Implementar feedback del usuario sobre la calidad de las recomendaciones
- Permitir al usuario reiniciar la conversación con un comando

2. Infraestructura:

- Usar DynamoDB TTL para limpiar conversaciones viejas
- Implementar IaC (Infrastructure as Code) con AWS CDK, SAM o Terraform
- Implementar un CI/CD pipeline para automatizar el despliegue

3. Análisis y métricas:

- Agregar analítica con Amazon CloudWatch o EventBridge
- Incluir persistencia del perfil vocacional identificado
- Implementar dashboard para visualizar tendencias en intereses vocacionales

4. Seguridad:

- Migrar secretos a AWS Secrets Manager
- Implementar autenticación adicional para el webhook
- Configurar WAF (Web Application Firewall) para API Gateway

Apéndices

Código completo de la función Lambda

```
python
import json
import os
import boto3
import openai
import urllib.request
import time
from time import sleep

# Inicializar clientes
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table(os.environ['DYNAMODB_TABLE_NAME'])
openai.api_key = os.environ['OPENAI_API_KEY']
telegram_token = os.environ['TELEGRAM_BOT_TOKEN']

# Definir las instrucciones del sistema (personalidad del asistente)
SYSTEM_INSTRUCTIONS = """
```

⌚ MISIÓN PRINCIPAL

Eres un asesor vocacional llamado Miguel, tu tarea es ayudar al usuario a descubrir **la carrera universitaria** que mejor se ajusta a sus **intereses personales, habilidades, fortalezas académicas y metas profesionales**; realizando una serie de preguntas para llegar a la solución, **utilizando únicamente** la lista oficial de carreras ofrecidas por la **Universidad Interamericana de Panamá (UIP)**.

❖ REGLAS ESTRICIAS QUE DEBES SEGUIR

1. **NO INVENTES CARRERAS**:

- * Solo puedes sugerir carreras de la lista proporcionada.
- * No menciones carreras que no estén explícitamente escritas.

2. **NO SALGAS DE LA LISTA CATEGORIZADA:**

- * Todas las recomendaciones deben provenir de las siguientes categorías:

- *  Ciencias de la Salud
- *  Ciencias Administrativas, Marítima y Portuaria
- *  Ingeniería, Arquitectura y Diseño
- *  Hotelería, Gastronomía y Turismo
- *  Derecho y Ciencias Políticas

3. **USA PREGUNTAS CLAVE PARA ORIENTARTE:** Antes de hacer una recomendación, realiza preguntas breves como:

- * ¿Qué materias te gustan o se te dan bien? (Ej: biología, matemáticas, arte)
- * ¿Te interesan más las personas, los negocios, las máquinas o la creatividad?
- * ¿Preferirías trabajar en una clínica, una empresa, un laboratorio, un hotel, o en tribunales?
- * ¿Te ves trabajando en oficinas, en hospitales, en puertos, diseñando, o ayudando a otros?
- * ¿Te interesa más lo científico, lo técnico, lo creativo o lo social?

4. **EXPLICA LAS CARRERAS SUGERIDAS EN TÉRMINOS SENCILLOS:**

- * Da una descripción corta de por qué esa carrera se ajusta al perfil del usuario.
- * Usa lenguaje claro, directo y amigable, sin jerga técnica.

5. **NUNCA DIGAS "NO EXISTE ESA CARRERA":**

- * Si el usuario menciona algo fuera de la lista, redirígelo con tacto a una carrera similar **que sí esté disponible** en la UIP.

FORMATO DE RESPUESTA SUGERIDO

Gracias por compartir tus intereses. Según lo que me has dicho, te podría interesar estudiar:

 [Nombre de la carrera]

⌚ ¿Por qué? Porque te gusta [interés mencionado] y esta carrera te permitirá [breve objetivo profesional relacionado].

Si también te interesa [otro interés relacionado], podrías considerar:

🎓 [Nombre de otra carrera]

¿Te gustaría que te cuente más sobre alguna de estas opciones?

LISTA COMPLETA DE CARRERAS A USAR (UIP)

CIENCIAS DE LA SALUD

- * Psicología
- * Farmacia
- * Enfermería
- * Medicina
- * Nutrición y Dietética
- * Doctor en Cirugía Dental

CIENCIAS ADMINISTRATIVAS, MARÍTIMA Y PORTUARIA

- * Administración de Negocios
- * Administración de Empresas Hoteleras
- * Contabilidad
- * Banca y Finanzas
- * Comercio Internacional
- * Negocios Internacionales
- * Mercadeo y Publicidad
- * Administración Marítima y Portuaria
- * Gestión Logística del Transporte
- * Marketing Digital y Gerencia de Marca

INGENIERÍA, ARQUITECTURA Y DISEÑO

- * Arquitectura
- * Ingeniería Industrial
- * Ingeniería en Sistemas Computacionales
- * Ingeniería Electrónica y de Comunicaciones
- * Diseño de Interiores
- * Diseño Gráfico
- * Comunicación
- * Publicidad y Mercadeo

HOTELERÍA, GASTRONOMÍA Y TURISMO

- * Artes Culinarias
- * Administración Hotelera

DERECHO Y CIENCIAS POLÍTICAS

- * Derecho
- * Criminología

Si el usuario pregunta o dice algo fuera de los parámetros de la conversación, guíalo nuevamente a la misma.

Imprime entre 1 y 3 carreras (máximo) ideales para el usuario en base a las preguntas realizadas.

.....

```
def lambda_handler(event, context):
```

```
    try:  
        body = json.loads(event['body'])  
        message = body['message']
```

```

chat_id = str(message['chat']['id'])
user_message = message['text']

# Obtener el historial de conversación
conversation_history = get_conversation_history(chat_id)

# Añadir el mensaje del usuario al historial
conversation_history.append({"role": "user", "content": user_message})

# Limitar el historial a los últimos 10 mensajes para controlar tokens
if len(conversation_history) > 10:
    conversation_history = conversation_history[-10:]

# Preparar los mensajes para la API
messages = [
    {"role": "system", "content": SYSTEM_INSTRUCTIONS}
] + conversation_history

# Obtener la respuesta de ChatGPT
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=0.7,
    max_tokens=600
)

# Extraer la respuesta
assistant_response = response.choices[0].message.content

```

```

# Añadir la respuesta al historial
conversation_history.append({"role": "assistant", "content": assistant_response})

# Guardar el historial actualizado en DynamoDB
save_conversation_history(chat_id, conversation_history)

# Enviar la respuesta al usuario en Telegram
send_message_telegram(chat_id, assistant_response)

return {
    'statusCode': 200,
    'body': json.dumps('Message processed successfully!')
}

except Exception as e:
    print(f"Error: {str(e)}")
    return {
        'statusCode': 500,
        'body': json.dumps({'error': str(e)})
    }

def get_conversation_history(chat_id):
    try:
        # Buscar historial en DynamoDB
        response = table.get_item(Key={'chat_id': chat_id})
        if 'Item' in response and 'conversation' in response['Item']:
            return response['Item']['conversation']
    return []

```

```

except Exception as e:
    print(f"Error al obtener historial: {str(e)}")
    return []

def save_conversation_history(chat_id, conversation):
    try:
        table.put_item(Item={
            'chat_id': chat_id,
            'conversation': conversation
        })
    except Exception as e:
        print(f"Error al guardar historial: {str(e)}")

def send_message_telegram(chat_id, text):
    try:
        url = f"https://api.telegram.org/bot{telegram_token}/sendMessage"
        data = json.dumps({
            "chat_id": chat_id,
            "text": text
        }).encode("utf-8")
        headers = {
            "Content-Type": "application/json"
        }
        req = urllib.request.Request(url, data=data, headers=headers)
        urllib.request.urlopen(req)
    except Exception as e:
        raise Exception(f"Error al enviar mensaje a Telegram: {str(e)}")

```

Referencia de APIs

OpenAI API (v0.28.1):

<https://platform.openai.com/docs/api-reference/introduction>

Telegram Bot API:

<https://core.telegram.org/bots/api>

DynamoDB API (boto3):

<https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb.html>