# CDW Documentation

## Azure Chatbot with OpenAI and Slack Integration 2

# Azure Chatbot with OpenAI and Slack Integration 2

This document provides instructions on how to deploy an application that uses OpenAI for a chatbot and Azure Bot to integrate into a Slack channel. This Wiki details an alternative way to integrate an Azure chatbot with Slack.

First, you will need to deploy the following infrastructure in Azure:

1. **Azure OpenAI**
    1. Once created, go into the Foundry portal and select Create New Deployment from a Base Model. This example uses a gpt-4.1 model with a deployment name of gpt-4.1
    2. After creating the deployment, go to Deployments on the left pane of the Foundry and select your deployment. Take note of the Target URI, Key, and Deployment Name. These values will be used later
    3. In AI Foundry, select Home from the left pane. Take note of the Azure OpenAI endpoint, this will also be used later. It should be in the following format: https://[app-name].openai.azure.com/
2. **Azure Bot**
    1. The bot should be deployed as a single tenant bot
    2. In Configuration on the left pane, take note of the Microsoft App ID and App Tenant ID. Then click Manage Password to be taken to the client secrets page of the app registration. Create a new client secret and take note of the value, as it will be inaccessible after leaving this page. If you leave the page and forget to note down the value, just create a new secret.
3. **App Service Plan**
    1. Nothing specific needs to be configured for this, but it is needed to deploy the App Service
4. **App Service**
    1. This example uses a Linux app with a runtime stack of Python 3.10. I'm not sure if it's accurate, but according to Copilot Python 3.13 is not supported at the time of writing this document.
    2. In the Overview pane of the App Service, take not of the Default domain (ends in .azurewebsites.net). This will be needed later
    3. Select Environment variables in the left pane and create the following environment variables:
        1. MicrosoftAppId: The App (Client) ID from the Azure Bot / App Registration
        2. MicrosoftAppPassword: The client secret that was created in the Azure Bot's app registration
        3. MicrosoftTenantId: The Tenant ID that was copied in the Azure Bot configuration
        4. OPENAI_API_KEY: The Key saved from the OpenAI deployment
        5. OPENAI_DEPLOYMENT: The name of the OpenAI deployment. By default it will be the same as the model name (gpt-4.1 in this example)
        6. OPENAI_ENDPOINT: The openai.azure.com endpoint you saved from AI Foundry. Make sure to remove the trailing slash (i.e. it should end in openai.azure.com, not openai.azure.com/)
    4. Select Configuration from the left pane. Ensure that the stack settings are set to Python 3.10, and that the startup command is “bash startup.txt”. Then scroll down and ensure that “Always on” is set to On

5. If you did not enable Application Insights during deployment, scroll down to Application Insights on the left pane and Enable it
6. Select App Service Logs on the left pane and set it to File System

5. **Key Vault** (not used in this walkthrough, but highly recommended for additional security)

### Azure Bot and Slack App

1. Go to https://api.slack.com/ and create a new app. Since there are restrictions on adding bots to CDW's Slack workspace, you will likely need to create your own personal workspace first using a free trial
2. Once the app is created, note down the Client ID, Client Secret, and Signing Secret
3. Return to your Azure Bot and go to Channels on the left pane. Select Slack from the list of Available Channels and fill in the information you saved from the Slack app. Also copy the two URLs the Bot gives you. The should be https://slack.botframework.com and https://slack.botframework.com/api/Events/[your-bot-name]
4. Apply the changes to add the Slack channel to your bot. We will come back here later for troubleshooting and validation
5. Go to Configuration on the left pane of the Azure Bot. For Messaging Endpoint, add the Default domain of your App Service with the “/api/messages” route appended to the end of it. It should look somewhat similar to the following: https://[app-name-and-other characters].[region].azurewebsites.net/api/messages. Apply the change
6. Return to your Slack app and make the following changes:
    1. In the App Home tab, scroll down and ensure “Messages Tab” is enabled, along with the checkbox for “Allow users to send Slash commands and messages from the messages tab”
    2. In the OAuth & Permissions tab, add the Redirect URL provided by the Azure Bot to the Redirect URLs section (should be https://slack.botframework.com)
    3. Also in the OAuth & Permissions tab, add the following scopes:
        1. app_mentions:read
        2. channels:history
        3. chat:write
        4. im:history
    4. In the Event Subscriptions tab, click the button to Enable Events, and paste the Request URL given by the Azure Bot (should be https://slack.botframework.com/api/Events/[your-bot-name])
    5. Also in the Event Subscriptions tab, expand the Subscribe to bot events section and add app_mention, message.channels, and message.im

### Backend Files and App Deployment

1. In VS Code (or whatever file editor / terminal you prefer), open a project in an empty folder. Create files named app.py, requirements.txt, and startup.txt. The contents of these files can be found at the end of this document
2. Notes on app.py:
    1. For troubleshooting purposes and since this is a test project, app.py prints the values of the App Service environment variables. THIS IS NOT SECURE. For additional training and to follow best practices, it is recommended that these print statements are removed and that the environment variables are stored and retrieved in an Azure Key Vault. This example is a bare-bones solution that does not follow best practice

2. Around line 87, you should see the following: url = f“{OPENAI_ENDPOINT}/openai/deployments/{OPENAI_DEPLOYMENT}/chat/completions?api-version=2025-01-01-preview”. You will want to compare this to the Target URI of your OpenAI deployment that you saved earlier. Make sure that the version at the end of the URL matches that of your deployment, and that the URL is the same when substituting in your environment variables. This is also why the trialing slash should have been removed from your OPENAI_ENDPOINT variable. If it was not, you will have to slashes in your URL and it will not work
3. In addition to the /api/messages route that will be used by the Azure Bot to communicate to your OpenAI deployment, app.py also creates a /health route to allow you to validate that your route is up. Once the app is deployed, you may open a web browser and navigate to your default domain with /health appended to the URL, and the browser should say “OK” if the site is up and running. The full link should look something like this: https://[app-name-and-other characters].[region].azurewebsites.net/health

3. Once all three files are created and saved in the folder, create a zip file of that folder. Ensure that all files are at the root level of the folder, not within any subfolders. Otherwise your app will not be able to find them. In PowerShell, if your terminal is in the correct folder, you can zip the folder using this command (mockapp.zip can be changed to whatever you want your zip file to be named): “Compress-Archive -Path * -DestinationPath mockapp.zip -Force”
4. Once the zip file is created, this is the PowerShell command that can be run to deploy the zip file to the web app (make sure to replace the words in brackets with your own information: “az webapp deployment source config-zip –resource-group [RG-NAME] –name [APP-SERVICE-NAME] –src [ZIP-FILE-NAME]”
5. When deploying the app, it is helpful to have the App Service Log Stream up to assist in identifying and troubleshooting errors. This can be done by going to your App Service in the Azure Portal and selecting Log stream on the left pane.

**Troubleshooting Tips**

1. The App Service log stream will be your best friend when trying to identify errors. That is where you will get the vast majority of your information when your deployment is not working properly
2. Another very helpful tool for testing is the Test in Web Chat option in Azure Bot. If you aren't sure if the problem lies between Slack and your Bot, or your Bot and App Service, this is a great way to test it. The Test in Web Chat option allows you to test the communication with your chatbot just like you would through Slack but going directly through the Azure Bot. This way you can validate that things are working between your Bot and App Service before moving on to troubleshooting the configuration between your Bot and Slack
3. If, when troubleshooting, you are instructed to update the startup.txt instructions to use gunicorn, or to forgo using startup.txt altogether and start the app directly from the App Service, I would recommend against that and making the files and startup command work as they are. In my testing, gunicorn caused many problems that I was not able to resolve with this method of implementation. And when trying to omit startup.txt, the App Service would ignore my requirements.txt file and therefore not download the necessary packages to run app.py. There are likely ways to make either of these options work, but I spent a lot of time trying both and was not able to find a solution
4. If you are going down a troubleshooting rabbit hole without making much progress, I highly recommend doing occasional sanity checks to make sure some of the basics are working. One of the main ways I did this was by adding many print statements to my code so that I would know exactly what was being executed and what wasn't, which made it easier to see where the issues were occurring. The other main thing I did was to simplify the solution to make sure

certain parts worked. That is why the /health route is there, just to make sure the app is deploying and running. I would also remove the OpenAI logic at certain points and replace it with an echo, so that the “chatbot” would just repeat back what I said. With that and using the Test in Web Chat option, I was able to eliminate a lot of variables and just focus on the connection between the App Service and the Azure Bot

## app.py

```
import os
import traceback
import logging
import requests
from aiohttp import web
from botbuilder.core import (
    BotFrameworkAdapterSettings,
    BotFrameworkAdapter,
    TurnContext,
)
from botbuilder.schema import Activity
from botframework.connector.auth import MicrosoftAppCredentials
from botframework.connector.aio import ConnectorClient
from types import MethodType

# Load environment variables
try:
    from dotenv import load_dotenv
    load_dotenv()
except ImportError:
    pass

APP_ID = os.getenv("MicrosoftAppId", "")
APP_PASSWORD = os.getenv("MicrosoftAppPassword", "")
TENANT_ID = os.getenv("MicrosoftTenantId", "")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "")
OPENAI_ENDPOINT = os.getenv("OPENAI_ENDPOINT", "")
OPENAI_DEPLOYMENT = os.getenv("OPENAI_DEPLOYMENT", "")

print(f"App ID: {APP_ID}")
print(f"Password: {APP_PASSWORD}")
print(f"Tenant ID: {TENANT_ID}")
print(f"OpenAI Endpoint: {OPENAI_ENDPOINT}")
print(f"OpenAI Deployment: {OPENAI_DEPLOYMENT}")

# Custom credentials class
class CustomAppCredentials(MicrosoftAppCredentials):
    def __init__(self, app_id, app_password):
        super().__init__(app_id, app_password)
```

```
        self.app_id = app_id
        self.app_password = app_password

    def signed_session(self, session=None):
        token_url =
f"https://login.microsoftonline.com/{TENANT_ID}/oauth2/v2.0/token"
        data = {
            "grant_type": "client_credentials",
            "client_id": self.app_id,
            "client_secret": self.app_password,
            "scope": "https://api.botframework.com/.default"
        }
        response = requests.post(token_url, data=data)
        result = response.json()
        if "access_token" not in result:
            raise Exception(f"Token request failed: {result}")
        token = result["access_token"]

        if session is None:
            session = requests.Session()
        session.headers.update({"Authorization": f"Bearer {token}"})
        return session

# Adapter setup
SETTINGS = BotFrameworkAdapterSettings(APP_ID, APP_PASSWORD)
ADAPTER = BotFrameworkAdapter(SETTINGS)

# Corrected connector client creation function
async def custom_create_connector_client(self, service_url: str,
identity=None, audience: str = None) -> ConnectorClient:
    print(f"□ Creating connector client for {service_url}")
    creds = CustomAppCredentials(APP_ID, APP_PASSWORD)
    return ConnectorClient(creds, base_url=service_url)

# Bind the method to the adapter
ADAPTER.create_connector_client = MethodType(custom_create_connector_client,
ADAPTER)

# Function to call OpenAI chat completion
def get_openai_response(user_input):
    headers = {
        "api-key": OPENAI_API_KEY,
        "Content-Type": "application/json"
    }
    payload = {
        "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": user_input}
        ]
    }
    url =
```

```
f"{OPENAI_ENDPOINT}/openai/deployments/{OPENAI_DEPLOYMENT}/chat/completions?
api-version=2025-01-01-preview"
    response = requests.post(url, headers=headers, json=payload)

    print(f"🔍 OpenAI API status code: {response.status_code}")
    try:
        result = response.json()
        print("🔍 OpenAI API response:")
        print(result)
        if "choices" not in result:
            raise Exception(f"Missing 'choices' in OpenAI response:
{result}")
        return result["choices"][0]["message"]["content"]
    except Exception as e:
        raise Exception(f"Error parsing OpenAI response: {str(e)}")


# Message handler
async def handle_message(turn_context: TurnContext):
    if turn_context.activity.type == "message":
        user_input = turn_context.activity.text
        try:
            reply = get_openai_response(user_input)
        except Exception as e:
            reply = f"Error calling OpenAI: {str(e)}"
        await turn_context.send_activity(reply)
    else:
        await turn_context.send_activity(f"[{turn_context.activity.type}]
event received")

# Endpoint
async def messages(req: web.Request) -> web.Response:
    print("=== Incoming request to /api/messages ===")
    print("🔍 Received request at /api/messages")
    try:
        body = await req.json()
        print(f"🔍 Payload: {body}")
        activity = Activity().deserialize(body)
        auth_header = req.headers.get("Authorization", "")
        await ADAPTER.process_activity(activity, auth_header,
handle_message)
        return web.Response(status=200)
    except Exception as e:
        print("🔍 Exception handling activity:")
        print(traceback.format_exc())
        return web.Response(status=500, text=f"Error: {str(e)}")

async def healthcheck(req: web.Request) -> web.Response:
    return web.Response(text="OK", status=200)

# App setup
```

```
APP = web.Application()
print("□ APP object created")
APP.router.add_post("/api/messages", messages)
APP.router.add_get("/health", healthcheck)

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Gunicorn entry point
async def init_app():
    print("□ init_app() starting full app")
    logger.info("□ init_app() starting full app")
    return APP

if __name__ == "__main__":
    print("□ __main__ is running")
    web.run_app(init_app(), host="0.0.0.0", port=int(os.environ.get("PORT",
8000)))
```

## requirements.txt

```
aiohttp
python-dotenv
botbuilder-core
botbuilder-schema
botframework-connector
```

## startup.txt

```
#!/bin/bash
echo "□ Running startup.txt"
pip install -r requirements.txt
exec python -u app.py
```

| Date | Performer | Type(Initial/Change/Review) | Overview of change |
|---|---|---|---|
| 07/17/2025 | Zach Hein | Initial | Initial Document |
| xx/xx/xx | xxxxx xxxx | Change | Changed information on document review process |
| xx/xx/xx | xxxxx xxxx | Review | 6 month document review |

AI Knowledge