A Principled Technologies report: Hands-on testing. Real-world results.

Build an Azure OpenAI application using your own enterprise data

We built an intelligent AI chat application using Microsoft Azure Cosmos DB and Azure AI Services

Decision-makers across many industries and business sizes are excited about the potential power of a customized chat service using OpenAI to solve a wide range of business problems. Maybe your goal is reducing customer support costs or helping your employees access or generate information more easily. Many organizations in e-commerce, financial services, and beyond are also using Azure Cosmos DB to store corporate data and perform transactions on that data. Those groups can meet their AI needs by building an Azure AI application that draws from Azure Cosmos DB, uses Azure AI Search to categorize your data, and uses Azure's access to OpenAI generative pre-trained transformer (GPT) models to process prompts and return completions with responses relevant to your own data. Not only can Azure Cosmos DB store a large amount of unstructured data related to the primary application, it can store information related to an AI chatbot conversation history, helping to provide later business value.

Azure Cosmos DB is a NoSQL distributed database designed by Microsoft to be operational in nature, enabling low latency responses and powering transaction-driven workloads and also intelligent, AI-powered applications. Unlike a traditional relational model database, Azure Cosmos DB can store unstructured data. By using Azure Cosmos DB to host your data, utilizing Azure AI Search to search and retrieve data relevant to a query, and providing a chat interface to Azure OpenAI, it is possible to set up a retrieval augmented generation (RAG) architecture and avoid having to train the model itself, all while protecting your data.

To demonstrate this, Microsoft engineering developed a sample Vector Search & AI Assistant application using a bicycle shop website as a sample business. Using its web-based chat interface, hypothetical customers of the bicycle shop could gather information on products by asking a sequence of connected questions that the solution uses to deliver better responses.[1] This method improves on the usual search function, which answers one question at a time and cannot link questions. This web-based application stores a customer's questions during a session, uses the questions to provide better prompts to Azure OpenAI, and uses Azure AI Search to act as a retrieval system for the business data stored in Azure Cosmos DB. This application was a good start, but it used a very small database. While Azure Cosmos DB is designed for production use in processing orders or other transactional workloads, this sample application did not have those e-commerce features.

To address those limitations, Principled Technologies (PT) extended this application by using a custom, much larger Azure Cosmos DB dataset. See the science behind the report for more details on how we extended the application, and read on to learn how you could benefit from building such an application yourself.

Build an Azure OpenAI application using your own enterprise data February 2024

The science behind the report:

Build an Azure OpenAI application using your own enterprise data

This document describes what we tested, how we tested, and what we found. To learn how these facts translate into real-world benefits, read the report Build an Azure OpenAI application using your own enterprise data.

We concluded our hands-on testing on November 11, 2023. During testing, we determined the appropriate hardware and software configurations and applied updates as they became available. The results in this report reflect configurations that we finalized on November 11, 2023 or earlier. Unavoidably, these configurations may not represent the latest versions available when this report appears.

How we tested

We created a set of five Azure VMs and one Azure Cosmos DB container in its own resource in the same region in which we deployed the AI-assisted chatbot. We used one VM to deploy the chatbot and four VMs as testing platforms. The container held a copy of the property-listings data. At deployment, the script copied the data into the application's Azure Cosmos DB dataset, and once the chatbot started, one chatbot process sent the data to the Azure AI Search vector database for ingest and indexing.

Requesting access to Azure Open AI Service

Before you can use Azure OpenAI Service, you must request access with Azure. You must also choose an Azure region in which to run the chatbot. This region must support Azure OpenAI text services. See "Azure OpenAI Service models" at https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models for a list of each region's capabilities.

1. Go to the form Request Access to Azure OpenAI Service at https://aka.ms/oai/access.
2. Provide your company details, subscription, and account information.
3. Select the OpenAI feature Text and code models.
4. Azure will send notification of approval by email.

Installing prerequisites for deployment and testing of the AI-assisted chatbot

1. Create the Azure resource group for deployment and testing:
    a. In the Azure Portal, click Resource groups.
    b. Click Create.
    c. Enter the group name, and select the region.
    d. Click Review + create.
    e. Click Create.
2. Create the Azure Cosmos DB account and database for the local copy of the property-listing data:
    a. In the Azure portal, go to the new resource group's dashboard, and click Create.
    b. Select Azure Cosmos DB, and click Create.
    c. On the Azure Cosmos DB for NoSQL panel, click Create.
    d. Enter the Account name, and select the Location.
    e. Click Review + create.

3. Create a VM set for the deployment and testing VMs.
   a. In the Azure portal, go to the new resource group's dashboard, and click Create.
   b. On the Virtual machine scale set panel, click Create.
   c. Enter the scale-set's name and region.
   d. For the image, select Ubuntu Server 22.04 LTS -x64 Gen2.
   e. For the size, select Standard_D4s_v3.
   f. For the authentication type, select Select SSH public key.
   g. Select the Scaling tab.
   h. For the Initial instance count, enter 5.
   i. Select Manual for Scaling policy.
   j. Click Review + Create.
   k. Click Create.
   l. Click Download private key and create resource.
4. Get each VM's public IP address.
5. On each VM, perform the following tasks:
   a. Log onto account `azureuser` using the SSH private key.
   b. Copy the private key into the file ~/.ssh/key.pem.
   c. Configure the account to use this key:

```
chmod 600 ~/.ssh/key.pem
mkdir ~/.ssh/controlmasters
chmod 700 ~/.ssh/controlmasters
cat <<EOF >> ~/.ssh/config
Host *
     IdentityFile ~/.ssh/key.pem
     HashKnownHosts yes
     StrictHostKeyChecking no
     ControlPath ~/.ssh/controlmasters/%r@%h:%p
     ControlMaster auto
     ControlPersist 2h
     ForwardX11 no
EOF
```

   d. Update Ubuntu:

```
sudo apt update
sudo apt -y upgrade
```

   e. Add the IP addresses of the VMs to the system hosts file:

```
cat <<EOF >> /etc/hosts
# VMs
IP_1     test01
IP_2     test02
IP_3     test03
IP_4     test04
IP_5     deploy
EOF
```

   f. Log onto each VM once to populate the SSH known-host file:

```
for i  in deploy test{01..04}; do
   echo $i
   ssh $i date
done
```

   g. Reboot the VM:

```
shutdown -r now
```

6. Log onto the deployment VM.
    a. Install prerequisite Ubuntu packages:

```
sudo apt-get install ca-certificates curl gnupg lsb-release wget
```

    b. Install PowerShell:

```
wget https://github.com/PowerShell/PowerShell/releases/download/v7.3.10/
powershell_7.3.10-1.deb_amd64.deb
sudo dpkg -i powershell_7.3.10-1.deb_amd64.deb
rm powershell_7.3.10-1.deb_amd64.deb
```

    c. Install the Azure command line tool:

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -sLS https://packages.microsoft.com/keys/microsoft.asc | \
  sudo gpg --dearmor -o /etc/apt/keyrings/microsoft.gpg
  sudo chmod go+r /etc/apt/keyrings/microsoft.gpg
AZ_DIST=$(lsb_release -cs)
cat <<EOF | sudo tee /etc/apt/sources.list.d/ azure-cli.list > /dev/null
  deb [arch='dpkg --print-architecture' signed-by=/etc/apt/keyrings/microsoft.gpg] \
  https://packages.microsoft.com/repos/azure-cli/ $AZ_DIST main
EOF
sudo apt update
sudo apt install azure-cli
```

    d. Add the prerequisite Azure packages:

```
for i in aks-preview application-insights storage-preview containerapp; do
  echo $i
  az extension add --name $i
az extension update --name $i
done
```

    e. Install docker per its documentation:

```
sudo apt remove docker.io docker-doc docker-compose docker-compose-v2 \
   podman-docker containerd runc
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
   sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod go+r /etc/apt/keyrings/docker.gpg
cat <<EOF | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
  deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
   https://download.docker.com/linux/ubuntu \
   $(. /etc/os-release && echo "$VERSION_CODENAME") stable
EOF
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin \
   docker-compose-plugin
sudo usermod -aG docker $USER
newgrp docker
```

    f. Install helm:

```
wget https://get.helm.sh/helm-v3.13.2-linux-amd64.tar.gz
tar -xf helm-v3.13.2-linux-amd64.tar.gz
sudo install linux-amd64/helm /usr/local/bin
rm -rf linux-amd64
```

7. For each testing VM:
   a. Log into the system.
   b. Install prerequisite Ubuntu packages:

```
sudo apt update
sudo apt install libx11-xcb1 libxcomposite1 libasound2 libatk1.0-0 libatk-bridge2.0-0 libcairo2
libcups2 libdbus-1-3 libexpat1 libfontconfig1 libgbm1 libgcc1 libglib2.0-0 libgtk-3-0 libnspr4
libpango-1.0-0 libpangocairo-1.0-0 libstdc++6 libx11-6 libx11-xcb1 libxcb1 libxcomposite1
libxcursor1 libxdamage1 libxext6 libxfixes3 libxi6 libxrandr2 libxrender1 libxss1 libxtst6
```

   c. Create a testing directory:

```
mkdir testing
```

   d. Install node and nodejs packages:

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || \
  printf %s "${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
nvm install 20.10.0
nvm install-latest-npm

npm i puppeteer
```

Downloading the AI-assisted chatbot and data to the deployment VM

1. Log into the deployment VM.
2. Clone the chatbot repository to get the chatbot code, configuration files, and deployment scripts:

```
git clone https://github.com/PrincipledTechnologiesInc/PT-AI-chatbot-with-Azure-Cosmos-DB-and-Azure-
AI-Search.git chatbot
cd chatbot
```

3. Download and uncompress the property-list data to a local file:

```
wget https://github.com/PrincipledTechnologiesInc/PT-AI-chatbot-with-Azure-Cosmos-DB-and-Azure-AI-
Search/raw/main/data/prop.js.gz
gunzip prop.js.gz
```

4. Obtain the connection string for your Azure Cosmos DB account:
   a. Go to the Azure portal.
   b. Click Resource groups, and select the Azure Cosmos DB account you plan to use to hold the data.
   c. In the left column, click Keys.
   d. Copy the PRIMARY CONNECTION STRING.
5. Create the dMT configuration file, `migration.json` (see below). Be sure to modify four values with "<<<>>>" in them as follows: In the sink section, change `FilePath` to use the full path to the `prop`.json file. In the source section, change the `ConnectionString`, `Database` name, and `Container` name to use yours.

   File: `migration.json`

```
{
  "Source": "JSON",
  "Sink": "Cosmos-nosql",
  "Operations": [
    {
      "SourceSettings": {
        "FilePath": "/ <<>>> /chatbot/data/prop.json"
      },
      "SinkSettings": {
        "ConnectionString": "AccountEndpoint= <<>>> ==",
        "Database":"<<>>> database",
        "Container":"<<<>>> properties",
        "PartitionKeyPath":"/name",
        "RecreateContainer": false,
        "BatchSize": 100,
        "ConnectionMode": "Direct",
        "MaxRetryCount": 5,
        "InitialRetryDurationMs": 200,
        "CreatedContainerMaxThroughput": 1000,
        "UseAutoscaleForCreatedContainer": true,
        "WriteMode": "InsertStream",
        "IsServerlessAccount": false
        }
    }
  ]
}
```

6. Upload the data to your data container:

```
./dmt
```

Deploying the AI-assisted chatbot

To deploy the chatbot, you will need the name of your Azure subscription and the Azure region where you have enabled Azure OpenAI Service. Running one script will create the necessary infrastructure, create, and deploy the chatbot.

As soon as the Azure API gateway starts, it will begin adding your data to the vector database. Depending on the size of your data, this conversion could take several hours.

1. Log into the deployment VM.
2. Change the source of the property-listings data to your container's endpoint. Change the ConnectionString to the one you obtained in step 4 in the section Downloading the AI-assisted chatbot and data to the deployment VM.
3. Start PowerShell in the chatbot directory:

```
pwsh
```

4. Start the chatbot deployment, which will create a new Azure resource group with the name you give:

```
./scripts/Unified-Deploy.ps1 -resourceGroup <GROUP> -location <REGION> -subscription <SUBSCRIPTION>
```

5. The script's final output is the URL for the chatbot's web front-end. You can also find the URL in the resource group created by running the following command, using the names of the resource GROUP and newly created AKS resource in that resource group:

```
az aks show -n <AKS> -g <GROUP> -o tsv --query addonProfiles.httpApplicationRouting.config.
HTTPApplicationRoutingZoneName
```

6. To determine whether the data-conversion process has finished, you have two choices.
   - Look for an upload completed message in the log for the Azure API gateway pod with the kubectl command; viz.,

```
# print the name of the API pod
kubectl get pods | awk '/ms-openai-cosmos-db-api-chat-service-web-api/ {print $1}'
kubectl logs API_POD | tail
```

   - Alternatively, you can check the number of entries in the vector database's index from the Azure AI Search dashboard. The number of indexed entries will be 444,121 when the conversions have ended.
     a. On the Azure portal, click the chatbot's resource group.
     b. Click the name of the Search service.
     c. On the right, click Indexes.
     d. The document count for the vector-index should be 444,121.

Running the performance testing

The directory `testing` contains data and scripts to automatically send questions to the chatbot from multiple users. The testing harness creates random questions by selecting a random entry from a file of templates and then replacing the variables in the template with entry randomly chosen from a file containing lists of countries, states, postal codes, amenities, etc.

We have not optimized the current chatbot architecture for more than 32 users, so we recommend using 32 or fewer users. The testing script will distribute the users over the four VMs.

1. Log onto the first testing VM, `test01`.
2. Go to the testing directory:

```
cd testing
```

3. Run the performance test for N users:

```
node test_prompts N
```

The results of the test will be in the file named "results_DateTime_NumUsers.txt". The file has four results for each user: the request's starting time, the elapsed time between the user's pressing enter and the app's printing the completion, the number of embedding tokens for the user's question, and the number of embedding tokens for the complete prompt and RAG context.

Read the report at https://facts.pt/2JHwrK6 ▶

This project was commissioned by Microsoft.

Principled Technologies®

Facts matter.®