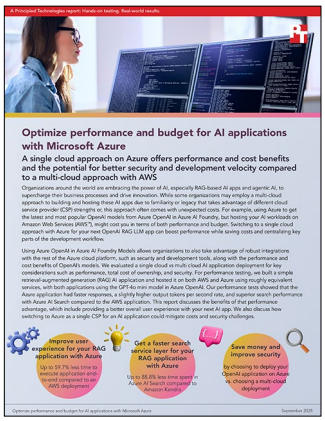

The science behind the report:

Optimize performance and budget for AI applications with Microsoft Azure

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 Optimize performance and budget for AI applications with Microsoft Azure.

We concluded our hands-on testing on June 10, 2025. 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 June 8, 2025 or earlier. Unavoidably, these configurations may not represent the latest versions available when this report appears.

Our results

To learn more about how we have calculated the wins in this report, go to http://facts.pt/calculating-and-highlighting-wins. Unless we state otherwise, we have followed the rules and principles we outline in that document.

Table 1: Results of our testing.

Measurement	Application location	1 user	50 users	100 users
Average time between tokens (ms)	Application running on Azure	6.32	7.51	7.64
	Application running on AWS	6.92	7.66	7.88
App elapsed time (ms)	Application running on Azure	1647.06	1527.90	1742.63
	Application running on AWS	2180.76	3794.85	4010.60
Average search elapsed time (ms)	Azure AI Search	557.24	297.44	505.01
	Amazon Kendra	1204.52	2659.64	2808.79

How we tested

This project implements a retrieval augmented LLM application using contextual search across two cloud providers. Both cloud providers used Azure OpenAI for AI services. The application simulates users submitting queries to a web app, processes the queries using a backend function, retrieves relevant context from a contextual search function, sends the query plus context to Azure OpenAI, and streams the query response back to the web UI via the function layer.

For services on Azure, we used Azure SQL Database, Azure AI Search, an Azure Function for backend processing, and an Azure Web App for the frontend interface.

For services on AWS, we used Amazon RDS with SQL Server, Amazon Kendra, Lambda for backend function processing, and an AWS Elastic Beanstalk web app for the frontend interface. We tested both providers' applications with client orchestration using Puppeteer, a lightweight browser load-testing tool.

Please contact PT for all scripts and code at info@principledtechnologies.com.

Azure infrastructure components overview

Table 2 provides an overview of the key components of the Azure infrastructure.

Table 2: Key components of the Azure infrastructure.

Azure service	Description	Level of service or other parameters			How we setup
Azure AI Search	Stores indexed documents with fields like chunk for text retrieval. Configured to support vector-based search with a text_vector field for similarity matching.	Service level: S1 Partition: 1 Region: US East 2			Terraform, then manual adjustment
		1-user	50-user	100-user	
		2 replicas	3 replicas	6 replicas	
Azure OpenAI service #1	Used for LLM	Deployment: gpt-4o-mini Deployment type: Regional Zone Provisioned Throughput, 50 PTUs Region: US East 2			Azure console
Azure OpenAI service #2	Used for embeddings and AI Search query vectorization	Deployment: text-embedding-ada-002 Deployment Type: Global Standard Quota: ~4M TPM			Azure console
Azure Function App + Azure App Service Plan	Hosts the backend logic for processing user queries and interacting with Azure AI Search and Azure OpenAI.	Instance: P1v3 (Linux) Python version: 3.12 Scale-out: Manual (see below)			Terraform, then manual adjustment
		1-user	50-user	100-user	
		3	6	10	
Azure Web App + Azure App Service Plan	Provides the frontend interface for users to submit queries and view results.	Instance: P1v3 (Linux) Startup parameters: gunicorn app:app --workers 4 --threads 10 --bind=0.0.0.0 Python version: 3.12 Scale-out: Manual (see below)			Terraform, then manual adjustment
		1-user	50-user	100-user	
		2	4	6	
Azure SQL Database	Used to hold the underlying data set, indexed by Azure AI Search	Standard S2 50 DTUs			Terraform
Azure Storage Account	Used for storing logs and metadata	ADLS Gen2			Hierarchical namespace enabled

High-level Azure workflow

The high-level workflow of deployment actions was as follows:

1. Deploy most items via Terraform.
2. Install Azure OpenAI service and model deployment via the Azure console.
3. Configure post-deployment options on each of the services where applicable.
4. Build and deploy Azure Function App and Azure Web App.
5. Load the sample dataset into Azure SQL Database, and index data with Azure AI Search.
6. Configure environment variables.
7. Configure client and Puppeteer scripts.

Deploying the Azure solution

Deploying Azure items via Terraform

We used Terraform to provision most of the Azure resources required for this project. Follow the steps below to configure the infrastructure components.

1. Install Terraform client based on instructions for your client here: https://developer.hashicorp.com/terraform/tutorials/azure-get-started/install-cli
2. Install the Azure CLI based on instructions for your client here: https://learn.microsoft.com/en-us/cli/azure/install-azure-cli
3. Download the necessary Terraform files.
4. Navigate to the directory where Terraform scripts are located.
5. Perform Terraform initialization:

```
terraform init
```

6. Log into the Azure CLI:

```
az login
```

7. Proceed to install the infrastructure components. You can install all at once, but we chose to install one by one, to ensure no errors. To install one by one, use the example scripts in the Terraform create script samples section below.

Configuring deployments of Azure OpenAI services for LLM and embeddings

Deploying the LLM

We configured a deployment of gpt-4o-mini for testing.

1. Create an Azure OpenAI resource in your resource group.
2. Navigate to your resource group in the Azure portal.
3. Click Create, search for Azure OpenAI, and choose Create.
4. Enter the name, region, and service level.
5. Click Next, and click Create.
6. In the Azure portal, go to the Overview page of the Azure OpenAI resource you created and click Explore Azure AI Foundry portal.
7. On the left menu, click Deployments.
8. Click Deploy model, Deploy base model.
9. Search for gpt-4o-mini, select it, and click Confirm.
10. For the Deployment type drop-down, select Regional Provisioned Throughput. Note that this will incur charges for the deployment being active.
11. To estimate your necessary PTUs, use the Input and Output TPM fields. If you know your PTUs you would like to use, use the slider bar to select the necessary PTU. For our testing we used 50 PTU.
12. Click Confirm Pricing, and acknowledge the warning.
13. Click Deploy.

Embedding deployment

We installed an additional Azure OpenAI service to be used for embeddings.

1. In the Azure Console, in the applicable resource group, click Create.
2. Search for Azure OpenAI, and click Create.
3. Change the region to US East 2, provide a unique name, and choose S0 pricing tier. Click Next.
4. For networking, allow all networks. We authenticated via API key. Click Next.
5. Add optional Tags if applicable. Click Next.
6. Click Create.
7. After the resource is added, go to the Overview page of the resource, and click Explore Azure AI Foundry portal.
8. On the left menu, click Deployments.
9. Click Deploy model, Deploy base model. Under the filter drop-down, select Embeddings.
10. Search for text-embedding-ada-002, select it, and click Confirm.
11. Click Customize, and adjust the TPM quota to approximately 1.5M TPM.
12. Click Deploy.

Configuring post-deployment options on each of the services where applicable

Configuring replicas on Azure AI Search

For testing under load, we noted that Azure AI Search required multiple replicas. By default, Azure AI Search has a single replica. For our testing, we varied the replica count by user count in our testing scenarios. To configure these, follow the steps below.

1. In the Azure Console, browse to your Azure AI Search service Overview page.
2. On the right, click the number next to Replicas.
3. Increase the slider to the necessary count.
4. Click Save.

Configuring the Python version on the Azure Functions app

We used Python 3.12. To configure the Python version on the Azure Functions app, follow the steps below.

1. In the Azure Console, browse to your Azure Functions app.
2. On the left menu, expand Settings, and choose Configuration.
3. Click General Settings.
4. Under Stack Settings, select Python version 3.12.

Configuring the Python version and threading on the Azure Web App

We used Python 3.12 on the web app and explicitly provided a startup parameter to optimize concurrency.

1. In the Azure Console, browse to your Azure Web App.
2. On the left menu, expand Settings, and choose Configuration.
3. Click General Settings.
4. Under Major version, be sure 3 is selected. Under Minor version, select Python version 3.12.
5. In the Startup command box, type the following:

```
gunicorn app:app --workers 4 --threads 10 --bind=0.0.0.0
```

6. Click Save.

Configuring network access on SQL Server

To load data and manage your SQL Server, adjust networking connectivity to allow access.

1. In the Azure Console, browse to your Azure SQL Database.
2. On the left menu, expand Security, and choose Networking.
3. Under Firewall rules, click Add your client IP address.
4. Be sure Allow Azure services and resources to access this server is enabled.
5. Click Save.

Creating an Azure Storage Table for logging results

We used an Azure Storage Table to log test results. To do so, follow the steps below.

1. In the Azure Console, browse to your Azure Storage Account.
2. Click Data Storage, and click Tables.
3. Click the plus sign to add a Table, provide a name, and click OK.

Loading the sample dataset and index data with Azure AI Search

Dataset information

We used a sample dataset containing musical instrument reviews from Kaggle, located at https://www.kaggle.com/datasets/eswarchandt/amazon-music-reviews . Download the dataset and convert from JSON to tabular form or CSV using any process you would like.

Loading data

We used SQL Server Management Studio to load our data.

1. Install SQL Server Management Studio if it is not already installed. Note that this is a Windows-only application.
2. Log into the Azure SQL Database, and browse to your database.
3. Right-click the Database, click Tasks, and choose Import Flat File.
4. Follow the remaining import wizard prompts to import the CSV data.

Indexing data in Azure AI Search

After the data is loaded in Azure SQL Database, index it in Azure AI Search using the following steps.

1. In the Azure console, navigate to your AI Search resource.
2. On the Overview page, click Import and vectorize data. This process will create the necessary embeddings to do vector queries.
3. Click Azure SQL Database.
4. Select the drop-downs associated with your data source.
5. Provide credentials, and click Next.
6. Select the column you wish to vectorize, the OpenAI resource you created for embeddings, and the deployment model (text-embedding-ada-002).
7. Click the acknowledgement button, and click Next.
8. Review final settings, and click Next.
9. Browse to the Indexer in AI Search, and review the status of the first Indexer run to ensure it was successful.

Building and deploying the Azure Function App and Azure Web App

Building the Azure Function App

You can deploy the Azure Function App using Azure Function core tools or VSCode. We provide steps below for using VSCode.

1. Open the project containing the code in VSCode.
2. Install the Azure extension if you have not already, and sign in.
3. Browse to your Subscription, and expand Function App. You should see your Azure Function App.
4. Right-click the Azure Function App and choose Deploy to Function App.
5. VSCode will prompt you for the directory if need be. Follow the on-screen instructions.

Building the Azure Web App

You can deploy the Azure Web App using VSCode.

1. Open the project containing the web app code in VSCode.
2. Install the Azure extension if you have not already, and sign in.
3. Browse to your Subscription, and expand App Services. You should see your Azure Web App.
4. Right-click the Azure Web App, in our case pt2025-orman-llm, and choose Deploy to Web App.
5. VSCode will prompt you for the directory if need be. Follow the on-screen instructions.

Configuring environment variables

There are several environment variables for both the Azure Function App and the Azure Web App in the code we used. We list those below with instructions on where to find the applicable variable or which value to use.

Azure Function App

- AZURE_COGNITIVE_SEARCH_ENDPOINT
  - Locate the value on the Azure AI Search Overview page, under URL. Copy this value, and use it.
- AZURE_COGNITIVE_SEARCH_NAME
  - Name of the Azure AI Search resource.
- AZURE_OPENAI_ENDPOINT
  - You must use the Endpoint provided within Azure AI Foundry, not the endpoint provided under Keys and Endpoint. To locate this endpoint, click Explore Azure AI Foundry, and click Deployments. Click your deployment, and click the Consume tab. Note the endpoint in the provided sample code. In our case, this was https://llm-pt-rag-app-eastus.openai.azure.com/openai/deployments/gpt-4o-mini .
- AZURE_OPENAI_KEY
  - To locate this Key, click Explore Azure AI Foundry, and select Deployments. Click your deployment, and click the Details tab. Copy the Key value.
- AZURE_OPENAI_MODEL
  - We used gpt-4o-mini.
- AZURE_SEARCH_INDEX
  - To locate this index name, browse to the Azure AI Search resource, select Search Management, and select Indexes. Copy the index string, and use it.
- AZURE_SEARCH_KEY
  - To locate this key value, browse to the Azure AI Search resource, select Settings, and click Keys. Copy the Primary admin key, and use it.
- AZURE_STORAGE_ACCOUNT_NAME
  - This storage account is used by the Azure Function for metadata. It should be set by the Terraform scripts included in the download package, but if not, set to the name of your Azure Storage Account.

Azure Web App

- AZURE_COSMOS_OR_TABLE
  - We initially wrote the application to log results to either (a) CosmosDB or (b) Azure Tables. To choose that option, provide either COSMOS or TABLE here. We chose TABLE.
- AZURE_FUNCTION_URL
  - This should contain the complete URL, include Function Key, to your Azure Function. To find this, browse to your Azure Function App. On the Overview page, click the Function (you will only see the Function here after deploying your code to Azure). Click the Function Keys tab, and copy the function. Retrieve the URL of your Function from the overview page. Then construct your full URL as follows:

```
https://FUNCTION-NAME-HERE.azurewebsites.net/api/HttpTriggerFunction?code=FUNCTION-KEY-VALUE-HERE
```

- AZURE_STORAGE_CONNECTION_STRING
  - To locate this connection string, browse to the Azure Storge Account resource, Security and networking, and click Access Keys. Beside the Connection string, click Show, and copy the connection string.
- AZURE_TABLE_NAME
  - This is the table name you created above. Browse to the Azure Storge Account resource, Data Storage, and then Tables. Note the table name you created to log resources.
- COSMOSDB_CONTAINER_NAME, COSMOSDB_DATABASE_NAME, COSMOSDB_ENDPOINT, COSMOSDB_KEY
  - The original code provided the option to log results to CosmosDB. If you wish to do so (instead of using Azure Tables), you can find this CosmosDB information under the Azure CosmosDB resource, should you choose to create it.

Azure Terraform create script examples

```
# Create resource group
terraform plan -target=azurerm_resource_group.rg
terraform apply -target=azurerm_resource_group.rg -auto-approve

# Create Azure SQL Database server
terraform plan -target=azurerm_mssql_server.sql_server
terraform apply -target=azurerm_mssql_server.sql_server -auto-approve

# Create database on Azure SQL Database server
terraform plan -target=azurerm_mssql_database.sql_database
terraform apply -target=azurerm_mssql_database.sql_database -auto-approve

# Create Azure AI Search resource
terraform plan -target=azurerm_search_service.cognitive_search
terraform apply -target=azurerm_search_service.cognitive_search -auto-approve

# Create Azure Storage account for Azure Function and logging results
terraform plan -target=azurerm_storage_account.function_storage
terraform apply -target=azurerm_storage_account.function_storage -auto-approve

# Create Azure App plan for Azure Function
terraform plan -target=azurerm_service_plan.function_plan
terraform apply -target=azurerm_service_plan.function_plan -auto-approve

# Create Azure Function App
terraform plan -target=azurerm_linux_function_app.function_app
terraform apply -target=azurerm_linux_function_app.function_app  -auto-approve

# Create Azure App plan for Azure Web App
terraform plan -target=azurerm_service_plan.llm-app-plan
terraform apply -target=azurerm_service_plan.llm-app-plan  -auto-approve

# Create Web App
terraform plan -target=azurerm_linux_web_app.llm-web-app
terraform apply -target=azurerm_linux_web_app.llm-web-app  -auto-approve
```

AWS infrastructure components overview

Table 3 provides an overview of the key components of the AWS infrastructure.

Table 3: Key components of the AWS infrastructure.

AWS service	Description	Level of service or other parameters			How we setup
Amazon Kendra	Stores indexed documents with fields like chunk for text retrieval. Varied query capacity by user count scenario. We used 10 QPS (+99 QCU) as this is the maximum supported for GenAI Enterprise Edition according to AWS support.	Edition: GenAI Enterprise Edition Region: US East 1			Terraform, then manual adjustment
		1-user	50-user	100-user	
		1 QPS (+9 QCU)	5 QPS (+49 QCU)	10 QPS (+99 QCU, max)	
Azure OpenAI service #1 (same OpenAI service used in Azure scenarios)	Used for LLM	Deployment: gpt-4o-mini Deployment type: Regional Zone Provisioned Throughput, 50 PTUs Region: Azure US East 2			Azure console
Lambda Function	Hosts the backend logic for processing user queries and interacting with Amazon Kendra and Azure OpenAI.	Allocated memory: 256MB Allocated storage: 512MB Python version: 3.12 Scale-out: Lambda scales by default			Terraform
AWS Elastic Beanstalk	Provides the frontend interface for users to submit queries and view results.	Type: Load balanced Instance type: t3.large (2 vCPU, 8GB RAM)			Terraform, then manual adjustment
		Procfile in app build: web: gunicorn app:app --workers 4 --threads 10 --bind=127.0.0.1:8000 Python version: 3.12 Scale-out: Manual (see below)			
		1-user	50-user	100-user	
		2min/2max	4min/4max	6min/6max	
Amazon RDS with SQL Server	Used to hold the underlying data set, indexed by Amazon Kendra	Edition: Express Instance class: t3.medium Allocated storage: 20GB Storage type: gp3			Terraform
Amazon DynamoDB	Used for storing logs and metadata	Default settings			Terraform

High-level AWS workflow

The high-level workflow of deployment actions on AWS were as follows:

1. Deploy most items via Terraform.
2. Check the Azure OpenAI service, and perform model deployment via the Azure console.
3. Configure post-deployment options on each of the services where applicable.
4. Load the sample dataset into Amazon RDS SQL Server and Index data with Amazon Kendra.
5. Build and deploy AWS Lambda function app and AWS Elastic Beanstalk app.
6. Configure environment variables.
7. Configure your client and Puppeteer scripts.

Deploying the AWS solution

One-time tasks

1. Ensure prerequisites are installed, namely Terraform, VS Code, and AWS CLI.
2. Create a Secret Key for Kendra to use for RDS access. Amazon Kendra needs access to RDS database resources to connect and create the data source and index. If you have not already created a Secret Key, do the following:
    a. Search for Secrets Manager.
    b. Click Store a new secret, and fill in the RDS details. IMPORTANT: To later appear in the necessary Kendra option drop-down, your secret name must begin with the prefix AmazonKendra-.

Building your environment with Terraform scripts

Prerequisites

Download Terraform, Lambda function code, and the web app code.

Editing Terraform files

1. Edit the database.tf file to include mgmt. Use the IP address for database management, and a strong password for the SQL Server admin user.
2. Change the management client IP in the section titled resource aws_security_group db_sg:

```
  ingress {
    description = "Allow SQL Server inbound traffic"
    from_port   = 1433
    to_port     = 1433
    protocol    = "tcp"
    cidr_blocks = ["123.123.123.123/32"] # Replace with your own IP
  }
```

3. Change the RDS password to be a strong unique password, and note the password:

```
resource "aws_db_instance" "sql_server" {
  identifier              = "sql-server-instance-name"
  engine                   = "sqlserver-ex"      # Express Edition
  instance_class           = "db.t3.medium"      # T3.medium instance
  allocated_storage        = 20                  # 20GB minimum for Express
  username                 = "admin"
  password                 = "strongpassword"
  skip_final_snapshot     = true
  publicly_accessible      = true
  multi_az                 = false                # Single AZ
  storage_type             = "gp3"
  vpc_security_group_ids   = [aws_security_group.db_sg.id]
  db_subnet_group_name     = aws_db_subnet_group.db_subnet_group.name
  backup_retention_period = 7
  tags = {
    ProjectName = "ProjectNameTag"
  }
}
```

4. Edit the kendra.tf terraform file to contain the edition of Kendra that you would like to use, and adjust the capacity units for query or documents:

```
resource "aws_kendra_index" "rag_index" {
  name        = "rag-kendra-index"
  edition     = "GEN_AI_ENTERPRISE_EDITION"
  role_arn    = aws_iam_role.kendra_execution_role.arn
  description = "Kendra index for RAG+LLM app"
  capacity_units {
    query_capacity_units   = 9
    storage_capacity_units = 0
  }
  tags = {
    ProjectName = "ProjectNameTag"
  }
}
```

Applying Terraform build

Run Terraform plan, and apply it to create Amazon Relational Database Service (RDS) SQL Server and associated resources, Kendra index and associated resources, and DynamoDB table.

1. Initialize your Terraform project by navigating to the project directory and running the following:

```
terraform init
```

2. Run terraform plan:

```
terraform plan
```

3. Run terraform apply:

```
terraform apply
```

This will result in the following:

- RDS
  - Creates VPC
  - Creates gateway
  - Creates route table
  - Creates two subnets
  - Associates subnets with route table
  - Creates security group that allows ingress from mgmt. machine and egress everywhere
  - Creates a subnet group, assigns two subnets above to the subnet group
  - Creates SQL Server RDS instance
- Kendra
  - Creates Kendra index
  - Creates Kendra execution role
  - Attaches a standard EC2 policy to that role
  - Adds a new policy to the role with certain permissions
  - Creates a security group to allow Kendra to access data in SQL Server that allows ingress from everywhere in VPC and egress from everywhere in VPC
  - Adds a security group rule of type ingress for 1433 for kendra to access SQL Server data
- DynamoDB
  - Create DynamoDB table

4. Verify in the AWS console that your RDS instance, Kendra index, and DynamoDB table are showing as ready and available.

Creating a SQL Server database creation and importing data

Creating the database

1. Gather these details from the AWS console and terraform script:
   - Endpoint (this will be the Server Name in SQL Server Management Studio)
   - Port (by default this is 1433)
   - User and password (use the credentials you provided in your terraform scripts)
2. On a Windows machine running SQL Server Management Studio, connect to the RDS SQL Server. Your SQL Server login user and password must match the AWS Secret Key that you created for Kendra to use.
3. Fill in server name, port, and check SQL Server Authentication.
4. Uncheck Trust Server Certificate, and click Connect.

5. Once connected, create a server login, database, and database user by running T-SQL below. Adjust database and log file sizes as necessary for your dataset.

```
USE [master]
GO
CREATE LOGIN [kendrauser] WITH PASSWORD=N'passwordhere', DEFAULT_DATABASE=[master], CHECK_
EXPIRATION=OFF, CHECK_POLICY=OFF
GO
CREATE DATABASE [kendradatabase]
 CONTAINMENT = NONE
 ON  PRIMARY
( NAME = N'kendradatabase', FILENAME = N'D:\rdsdbdata\DATA\kendradatabase.mdf' , SIZE = 5120KB ,
FILEGROWTH = 10%)
 LOG ON
( NAME = N'kendradatabase_log', FILENAME = N'D:\rdsdbdata\DATA\kendradatabase_log.ldf' , SIZE =
1024KB , FILEGROWTH = 10%)
GO
USE [kendradatabase]
GO
CREATE USER [kendrauser] FOR LOGIN [kendrauser] WITH DEFAULT_SCHEMA=[dbo]
GO
USE [kendradatabase]
GO
ALTER ROLE [db_datareader] ADD MEMBER [kendrauser]
GO
```

Importing data

1. Right-click database, choose Tasks, and select Import Data. You can import data from flat file or another SQL Server instance.

Configuring AWS Kendra (post creation)

Updating policy

1. Update the policy that your terraform created to ensure that to policy attached to the role that Kendra uses has the appropriate permissions.
2. After the Kendra index creation is completed, navigate to the Kendra index in the AWS console and note the index ID, relevant region (e.g. us-east-1), and your AWS account number.
3. Navigate to the applicable Role. In AWS console, search Roles (this will show under IAM).
4. On the Roles page, search for and click the role created via terraform, i.e., rag-kendra-execution-role.
5. Click the policy attached to the role by terraform, i.e., rag-kendra-policy.
6. Modify the following line (Your final edits should include the actual region, the actual AWS account number, and the actual Kendra index ID):

```
Resource = "arn:aws:kendra:region-here:aws-account-here:index/index-id-here"
```

Adding a custom index field

Add a custom field to the AWS Kendra index to capture the full text coming from the underlying SQL Server data field that we eventually want to search.

1. Search Kendra in the AWS search bar and navigate to the Kendra page.
2. Click your Kendra index.
3. On the left menu, click Facet definition.
4. In the Index field section, click Add Field.
5. Fill in the details for your additional field. In our sample application and sample data set, we matched the field name from our underlying data set
   - Field name: reviewText
   - Data type: String
   - Usage types: Select facetable, searchable, displayable
6. Click Add.

Creating the Kendra Index data source

This is the actual connection to your underlying data and the task that will invoke a sync.

1. In AWS, search for Kendra, and navigate to the Kendra page.
2. Click your Kendra index.
3. In the left menu, click Data sources.
4. Click Add Data sources.
5. Find the connector you wish to use. We used Amazon RDS (Microsoft SQL Server) connector. Click Add connector.
6. Fill in the following parameters for your data source connector.
   - Data Source Name: Identifier in the AWS console for this data source
   - Tag: Add tags as desired
   - Host: This should be what the RDS screen calls your Endpoint, e.g., sql-server1007.c40wvk2stayd.us-east-1.rds.amazonaws.com.
   - Port: 1433
   - Instance: This is NOT the SQL instance or DB Identifier. It should be the database name.
7. Uncheck Enable SSL Certificate Location.
8. Choose your AWS Secret that you created in Secrets Manager.
9. Choose the VPC created by the database terraform.
10. Choose the subnets created by the database terraform.
11. Choose the VPC security groups created by the database terraform.
12. Choose the IAM role created by the database kendra terraform.
13. Enter your SQL query. For our dataset (https://www.kaggle.com/datasets/eswarchandt/amazon-music-reviews) and schema, we used:

```
SELECT
id,
summary,
reviewText
FROM kendradatabase.dbo.MusicReviews
```

14. Enter field details:
    - Primary Key: id
    - Title: summary
    - Body: reviewText
    - Sync Mode: full sync
    - Schedule: run on demand
15. On the Field Mappings screen, choose Add Field. Use the following settings:
    - JDBC: reviewText
    - Index: reviewText
    - Data type: String
16. Click Add.
17. Click Next.
18. Review everything, and click Add Data Source.
19. Back on the data source summary page, choose Sync now. The sync may take some time. Check the status of the sync to resolve any errors that may appear.

Creating, deploying, and configuring the Lambda function

Creating the Lambda function

1. In the AWS search bar, enter Lambda, and navigate to the Lambda landing page.
2. Click Create Function.
3. Choose the following options:
   - Author from scratch
   - Type in function name, i.e. rag-llm-function
   - Choose the Runtime. For our application we used Python 3.12
   - Choose the architecture. For our application we used x86_64

4. Expand Change default execution role, and note the role name that will be created. It is a new role that will be prefixed by your function name. For example, if your function name is rag-llm-func, then the role that AWS creates would be in the format of rag-llm-func-role-abc123def56.
5. Click Create Function.

Adjusting Lambda permissions

When you create a Lambda function, by default AWS creates a new role prefixed with your function name and assigns permissions to it. For example, if your function name is rag-llm-func, then the role that AWS creates would be rag-llm-func- role-abc123def56. Follow the steps below to adjust the permissions for that function:

1. Gather the AWS region, AWS account number, and Kendra Index ID.
2. In the AWS search bar, search for Roles (under IAM), and click Roles.
3. In Roles, search for the function role, prefixed by your function name. Click the role.
4. Click Add permissions, and click Create inline policy.
5. Click JSON (instead of Visual).
6. Add this JSON to the policy, and insert your AWS region, account number, and Kendra Index ID.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
    "Effect": "Allow",
    "Action": [
      "kendra:Query"
    ],
    "Resource": "arn:aws:kendra:us-east-1:636009999999:index/12345abc-6789-abcd-85a8-67270937a767"
    }
  ]
}
```

7. Click Next.
8. Give the policy a name such as kendra-access-policy, and click Create Policy or Save.

Building and deploying the Lambda function

1. Install dependencies for the application to a folder.
2. In your terminal, navigate to the project's folder (the folder containing requirements.txt and handler.py).
3. Install dependencies you will need to package with your deployment bundle:

```
mkdir python
pip install -r requirements.txt -t python/
```

4. Zip your application folders to prepare for deployment. Follow these steps to gather both the dependencies and the application:

```
cd python
zip -r ../lambda_deploy.zip .
cd ..
zip -r lambda_deploy.zip handler.py shared/
```

5. In the AWS console, upload the zip file. Navigate to the Lambda function you created and on the Code tab, choose Upload from, and zip the file.
6. Click Upload, select your file, and click Save.

Adjusting the Lambda configuration

1. Set the handler name. In our case, this was a file named handler.
2. Navigate to your function in the AWS console.
3. On the Code tab, just below the Code source, under the Runtime settings section, click Edit.
4. For handler, modify the field to handler.lambda_handler.
5. Set Lambda environment variables.
6. Navigate to your Lambda function.

7. Click the Configuration tab.
8. Click the Environment variables menu.
9. Click Edit.
10. Click Add Environment variable for each item you wish to add. We added the following for our functioning code:
    - AZURE_OPENAI_API_VERSION: 2025-01-01-preview
    - AZURE_OPENAI_ENDPOINT: https://eastus2.api.cognitive.microsoft.com/openai/deployments/gpt-4o
    - AZURE_OPENAI_KEY: keyhere
    - AZURE_OPENAI_MODEL: gpt-4o-2024-11-20
    - KENDRA_INDEX_ID: indexidhere
    - K_RESULTS: 2
11. Adjust Lambda timeout and memory allocated.
12. Navigate to your Lambda function.
13. Click Configuration.
14. Click General Configuration.
15. Click Edit.
16. Modify the timeout value.
17. Modify the allocated memory. We set this to 256MB.
18. Modify the Ephemeral storage. We used 512MB.
19. Click Save.

Testing the Lambda function

1. Navigate to your Lambda function.
2. Click the Test tab.
3. Name your test event.
4. Enter JSON for the payload. Our example app used the following:

```
{
  "body": "{\"query\": \"Question for the LLM here?\"}"
}
```

5. Optionally click Save, and click Test.
6. Analyze results in the upper portion of the test window.

Deploying the Elastic Beanstalk (EB) application

Creating necessary roles and policies for EB

As part of the EB environment creation, you will select what it refers to as EC2 instance profile security settings. This simply refers to an IAM role that the EC2 instances use for permissions. Here, we pre-create that role for later use.

1. Create a policy.
2. Navigate to IAM, and click Policies.
3. Click Create policy, select JSON, and paste this JSON. Note that this contains the default recommended permissions, but we have also added permissions specific to our application, such as Lambda execution and DynamoDB put item permission. Be sure and change your region, AWS account number, DynamoDB table, and Lambda function name.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "BucketAccess",
            "Action": [
                "s3:Get*",
                "s3:List*",
                "s3:PutObject"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:s3:::elasticbeanstalk-*",
                "arn:aws:s3:::elasticbeanstalk-*/*"
```

```
            ]
        },
        {
            "Sid": "XRayAccess",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "CloudWatchLogsAccess",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogStream",
                "logs:DescribeLogStreams",
                "logs:DescribeLogGroups"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:logs:*:*:log-group:/aws/elasticbeanstalk*"
            ]
        },
        {
            "Sid": "ElasticBeanstalkHealthAccess",
            "Action": [
                "elasticbeanstalk:PutInstanceStatistics"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:elasticbeanstalk:*:*:application/*",
                "arn:aws:elasticbeanstalk:*:*:environment/*"
            ]
        },
        {
            "Sid": "AllowLambdaInvoke",
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-1:636009999999:function:rag-llm-func"
        },
        {
            "Sid": "AllowDynamoDBAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:63600999999:table/table-dynamodb"
        }
    ]
}
```

4. Name the policy with a relevant name, such as AWS-ElasticBeanstalk-EC2-policy.

Creating a Role

1. In IAM, click Roles.
2. Click Create Role.
3. Choose AWS Service, and select EC2 use case. Click Next.
4. On the add permissions page, search for and find the policy which you created above (AWS-ElasticBeanstalk-EC2-policy).
5. Name the role, e.g., AWS-EC2-ElasticBeanstalk-role, and click Create Role.

Creating the Elastic Beanstalk application

1. In AWS Search bar, search for Elastic, and click Elastic Beanstalk (EB).
2. Click Create application. This will start a series of wizard-based questions to answer that will in turn create roles, EC2 instances, scaling policies, load balancers, and so on. Many of these settings are configurable after the environment is created as well. Here, we document what we selected for this deployment, but each deployment may differ.

Configuring the environment

1. Choose Web server environment.
2. Name your application.
3. Name your EB environment.
4. Optionally provide a domain.
5. Choose Managed platform.
6. Select Python 3.12.
7. Choose Sample application. We will later deploy our application.
8. Choose Single instance. If choosing to scale, you will later select options that change this setting.
9. Configure service access. This screen provides your web application necessary permissions and roles to later adjust should your application need access to other services such as Lambda, DynamoDB, and so on.
10. If this is your first time creating a Lambda application, choose Create and use new service role. Otherwise, choose an existing Beanstalk service role.
11. Select a key pair to use for connecting to the VMs underlying the EB services. If you do not have a key pair, in a separate window, browse to key pairs in the AWS console and create a key pair, then refresh the EB menu.
12. For the EC2 instance profile, select the role you created above (e.g., AWS-EC2-ElasticBeanstalk-role).

Setting up networking, database, and tags

1. Leave VPC blank, public IP unchecked, and ensure that Enable database is toggled off.
2. Add tags as necessary.
3. Configure instance traffic and scaling.
4. Leave the Instances information section with defaults.
5. Under Capacity, select:

    - Load balanced
    - 1 for Min, 1 for Max (we later change this for certain scale scenarios)
    - On-demand instance
    - x86_64 architecture
    - Instance types – we chose t3.small for the first launch order, and t3.medium as the next launch order
    - Availability Zones: Any 1
    - Placement: us-east-1a
    - Scaling metric: CPU > 80% (scale up 1), CPU < 40% (scale down 1)
    - Period: 2
    - Breach duration: 1
    - Load balancer: application, dedicated

Configuring updates, monitoring, and logging

1. Select basic health monitoring.
2. Uncheck Managed updates.
3. Add an email for alerts (optional).
4. Under Rolling updates, leave all settings as default.
5. Under Platform software, leave all settings as default.

Reviewing EB configuration

1. Review all settings, and click Submit.
2. Monitor the EB creation using the Events tab.
3. Upload and deploy your application code.
4. Upload and deploy your application.
5. On your local machine in a terminal, navigate to the folder containing the EB application (the folder containing app.py).
6. Zip the files:

```
zip -r rag-llm-app.zip . -x "*.DS_Store" "__MACOSX"
```

7. In the AWS console, navigate to your EB environment. On events tab, click Upload and Deploy.
8. Choose your zip file, name it, and click Deploy.

9. Monitor progress from the Events tab.
10. Check and modify your EB environment variables. The upload package perhaps contained placeholder or older values for AWS region, Kendra ID, Lambda function name, or DynamoDB table. You must update this to be current based on your prior selections and named resources.
11. In the AWS console, navigate to your EB environment. From the left menu, click Configuration.
12. Under Updates, monitoring, and logging, click Edit.
13. Scroll down to Environment properties values. There should be several that populated from your source code upload. Update AWS region, Kendra ID, Lambda function name, and DynamoDB table as necessary for your current infrastructure.
14. Click Apply.
15. Monitor the Events page to be sure the EB environment applies your changes.

Restarting EB servers

1. In the AWS console, navigate to your EB environment.
2. Click Actions, and select Restart app server(s).
3. Wait for the process to complete with success.
4. Test the web application. Note you may need to add a certificate using AWS Certificate Manager and adjust settings using a domain you own. Check your application and if it is not working, use the Logs tab to request logs from applicable services.

Common parameters

Both our Azure and AWS application used a common python codebase, and we changed only the necessary components, for example switching our search module from Azure AI Search to Amazon Kendra search. Both applications shared several common parameters for search and Azure OpenAI:

- Search
  - We truncated context responses at 600 characters maximum
  - We used K=2 for both search services (top two results)
- Azure OpenAI
  - We prepended the context string with a random 10-character string to avoid prompt caching
  - Parameters
  - Output MAX_TOKENS = 400
  - TEMPERATURE = 1
  - TOP_P = 1
  - FREQUENCY_PENALTY = 0
  - PRESENCE_PENALTY = 0

General: Configuring your client and Puppeteer scripts

We used a two-socket, recent-generation server on which to run our client processes. To perform the test against the remote cloud resource we used Puppeteer version 24.7.0. You can install Puppeteer by following instructions at https://pptr.dev/guides/installation .

We then ran the pupp-test-2.js script located in the downloads.

We used the following user, question count, and wait periods for each test.

Table 4: Parameters we used during testing.

Parameter	1 concurrent user scenario	50 concurrent users scenario	100 concurrent users scenario
Max users	1	100	150
Concurrent users	1	50	100
Questions per user	100	70	70
User startup delay (s)	1	2	2
Question wait delay (s)	0.5	0.5	0.5

Read the report ▶

This project was commissioned by Microsoft.

Principled Technologies®

Facts matter.®