Create an Azure OpenAI, LangChain, ChromaDB, and Chainlit Chat App in Container Apps using Terraform

By Paolo Salvatori

Published Jul 27 2023 06:47 AM 17.8K Views

This article shows how to quickly build chat applications using Python and leveraging powerful technologies such as OpenAI ChatGPT models, Embedding models, LangChain framework, ChromaDB vector database, and Chainlit, an open-source Python package that is specifically designed to create user interfaces (UIs) for AI applications. These applications are hosted on Azure Container Apps, a fully managed environment that enables you to run microservices and containerized applications on a serverless platform.

- **Simple Chat**: This simple chat application utilizes OpenAI's language models to generate real-time completion responses.
- **Documents QA Chat**: This chat application goes beyond simple conversations. Users can upload up to 10 `.pdf` and `.docx` documents, which are then processed to create vector embeddings. These embeddings are stored in ChromaDB for efficient retrieval. Users can pose questions about the uploaded documents and view the Chain of Thought, enabling easy exploration of the reasoning process. The completion message contains links to the text chunks in the documents that were used as a source for the response.

Both applications use a user-defined managed identity to authenticate and authorize against Azure OpenAI Service (AOAI) and Azure Container Registry (ACR) and use Azure Private Endpoints to connect privately and securely to these services. The chat UIs are built using Chainlit, an open-source Python package designed

Skip to main content

explicitly for creating AI applications. Chainlit seamlessly integrates with LangChain, LlamaIndex, and LangFlow, making it a powerful tool for easily developing ChatGPT-like applications.
By following our example, you can quickly create sophisticated chat applications that utilize cutting-edge technologies, empowering users with intelligent conversational capabilities.

You can find the code and Visio diagrams in the companion GitHub repository. Also, check the following articles:

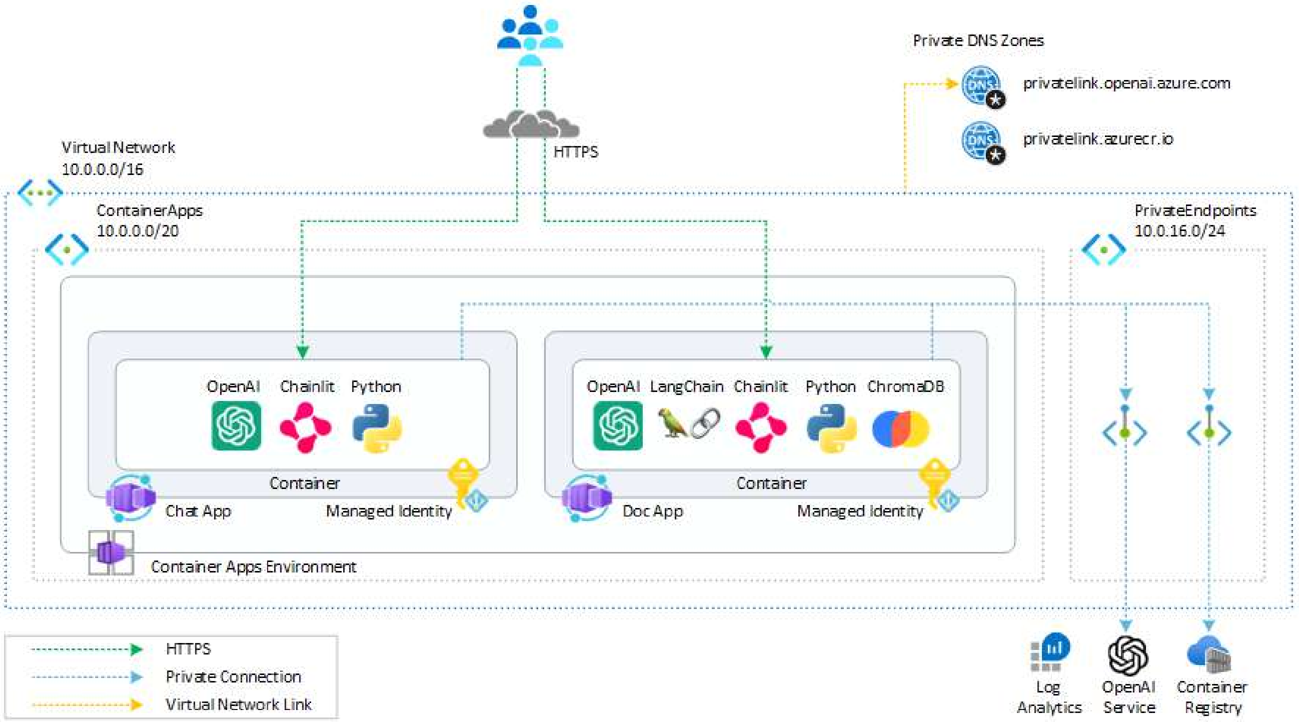
- Deploy and run an Azure OpenAI ChatGPT application on AKS via Bicep
- Deploy and run an Azure OpenAI ChatGPT application on AKS via Terraform

Prerequisites

- An active Azure subscription. If you don't have one, create a free Azure account before you begin.
- Visual Studio Code installed on one of the supported platforms along with the HashiCorp Terraform.
- Azure CLI version 2.49.0 or later installed. To install or upgrade, see Install Azure CLI.
- `aks-preview` Azure CLI extension of version 0.5.140 or later installed
- Terraform v1.5.2 or later.

Architecture

The following diagram shows the architecture and network topology of the sample:



This sample provides two sets of Terraform modules to deploy the infrastructure and the chat applications.

Infrastructure Terraform Modules

You can use the Terraform modules in the `terraform/infra` folder to deploy the infrastructure used by the sample, including the Azure Container Apps Environment, Azure OpenAI Service (AOAI), and Azure Container Registry (ACR), but not the Azure Container Apps (ACA). The Terraform modules in the `terraform/infra` folder deploy the following resources:

- azurerm_virtual_network: an Azure Virtual Network with two subnets:
  - `ContainerApps` : this subnet hosts the Azure Container Apps Environment.
  - `PrivateEndpoints` : this subnet contains the Azure Private Endpoints to the Azure OpenAI Service (AOAI) and Azure Container Registry (ACR) resources.
- azurerm_container_app_environment: the Azure Container Apps Environment hosting the Azure Container Apps.
- azurerm_cognitive_account: an Azure OpenAI Service (AOAI) with a GPT-3.5 model used by the chatbot applications. Azure OpenAI Service gives customers advanced language AI with OpenAI GPT-4, GPT-3, Codex, and DALL-E models with Azure's security and enterprise promise. Azure OpenAI co-develops the APIs with OpenAI, ensuring compatibility and a smooth transition from one to the other. The Terraform modules create the following models:
  - GPT-35: a `gpt-35-turbo-16k` model is used to generate human-like and engaging conversational responses.
  - Embeddings model: the `text-embedding-ada-002` model is to transform input documents into meaningful and compact numerical representations called embeddings. Embeddings capture the semantic or contextual information of the input data in a lower-dimensional space, making it easier for machine learning algorithms to process and analyze the data effectively. Embeddings can be stored in a vector database, such as ChromaDB or Facebook AI Similarity Search, explicitly designed for efficient storage, indexing, and retrieval of vector embeddings.
- azurerm_user_assigned_identity: a user-defined managed identity used by the chatbot applications to acquire a security token to call the Chat Completion API of the ChatGPT model provided by the Azure OpenAI Service and to call the Embedding model.
- azurerm_container_registry: an Azure Container Registry (ACR) to build, store, and manage container images and artifacts in a private registry for all container deployments. In this sample, the registry stores the container images of the two chat applications.
- azurerm_private_endpoint: an Azure Private Endpoint is created for each of the following resources:
  - Azure OpenAI Service (AOAI)
  - Azure Container Registry (ACR)
- azurerm_private_dns_zone: an Azure Private DNS Zone is created for each of the following resources:
  - Azure OpenAI Service (AOAI)
  - Azure Container Registry (ACR)
- azurerm_log_analytics_workspace: a centralized Azure Log Analytics workspace is used to collect the diagnostics logs and metrics from all the Azure resources:
  - Azure OpenAI Service (AOAI)
  - Azure Container Registry (ACR)
  - Azure Container Apps (ACA)

Application Terraform Modules

You can use these Terraform modules in the `terraform/apps` To deploy the Azure Container Apps (ACA) using the Docker container images stored in the Azure Container Registry you deployed in the previous step.

- azurerm_container_app: this sample deploys the following applications:
  - **chatapp**: this simple chat application utilizes OpenAI's language models to generate real-time completion responses.

Skip to primary navigation
Skip to main content

- **docapp**: This chat application goes beyond conversations. Users can upload up to 10 `.pdf` and `.docx` documents, which are then processed to create vector embeddings. These embeddings are stored in ChromaDB for efficient retrieval. Users can pose questions about the uploaded documents and view the Chain of Thought, enabling easy exploration of the reasoning process. The completion message contains links to the text chunks in the files that were used as a source for the response.

Azure Container Apps

Azure Container Apps (ACA) is a serverless compute service provided by Microsoft Azure that allows developers to easily deploy and manage containerized applications without the need to manage the underlying infrastructure. It provides a simplified and scalable solution for running applications in containers, leveraging the power and flexibility of the Azure ecosystem.

With Azure Container Apps, developers can package their applications into containers using popular containerization technologies such as Docker. These containers encapsulate the application and its dependencies, ensuring consistent execution across different environments.

Powered by Kubernetes and open-source technologies like Dapr, KEDA, and envoy, the service abstracts away the complexities of managing the infrastructure, including provisioning, scaling, and monitoring, allowing developers to focus solely on building and deploying their applications. Azure Container Apps handles automatic scaling, and load balancing, and natively integrates with other Azure services, such as Azure Monitor and Azure Container Registry (ACR), to provide a comprehensive and secure application deployment experience.

Azure Container Apps offers benefits such as rapid deployment, easy scalability, cost-efficiency, and seamless integration with other Azure services, making it an attractive choice for modern application development and deployment scenarios.

Azure OpenAI Service

The Azure OpenAI Service is a platform offered by Microsoft Azure that provides cognitive services powered by OpenAI models. One of the models available through this service is the ChatGPT model, which is designed for interactive conversational tasks. It allows developers to integrate natural language understanding and generation capabilities into their applications.

Azure OpenAI Service provides REST API access to OpenAI's powerful language models including the GPT-3, Codex and Embeddings model series. In addition, the new GPT-4 and ChatGPT model series have now reached general availability. These models can be easily adapted to your specific task, including but not limited to content generation, summarization, semantic search, and natural language-to-code translation. Users can access the service through REST APIs, Python SDK, or our web-based interface in the Azure OpenAI Studio.

You can use Embeddings model to transform raw data or inputs into meaningful and compact numerical representations called embeddings. Embeddings capture the semantic or contextual information of the input data in a lower-dimensional space, making it easier for machine learning algorithms to process and analyze the data effectively. Embeddings can be stored in a vector database, such as ChromaDB or Facebook AI Similarity Search (FAISS), explicitly designed for efficient storage, indexing, and retrieval of vector embeddings.

The Chat Completion API, which is part of the Azure OpenAI Service, provides a dedicated interface for interacting with the ChatGPT and GPT-4 models. This API is currently in preview and is the preferred method for accessing these models. The GPT-4 models can only be accessed through this API.

Skip to main content
Skip to primary navigation

GPT-3, GPT-3.5, and GPT-4 models from OpenAI are prompt-based. With prompt-based models, the user interacts with the model by entering a text prompt, to which the model responds with a text completion. This completion is the model's continuation of the input text. While these models are compelling, their behavior is also very sensitive to the prompt. This makes prompt construction a critical skill to develop. For more information, see Introduction to prompt engineering.

Prompt construction can be complex. In practice, the prompt acts to configure the model weights to complete the desired task, but it's more of an art than a science, often requiring experience and intuition to craft a successful prompt. The goal of this article is to help get you started with this learning process. It attempts to capture general concepts and patterns that apply to all GPT models. However, it's essential to understand that each model behaves differently, so the learnings may not apply equally to all models.

Prompt engineering refers to the process of creating instructions called prompts for Large Language Models (LLMs), such as OpenAI's ChatGPT. With the immense potential of LLMs to solve a wide range of tasks, leveraging prompt engineering can empower us to save significant time and facilitate the development of impressive applications. It holds the key to unleashing the full capabilities of these huge models, transforming how we interact and benefit from them. For more information, see Prompt engineering techniques.

Vector Databases

A vector database is a specialized database that goes beyond traditional storage by organizing information to simplify the search for similar items. Instead of merely storing words or numbers, it leverages vector embeddings - unique numerical representations of data. These embeddings capture meaning, context, and relationships. For instance, words are represented as vectors, whereas similar words have similar vector values.

The applications of vector databases are numerous and powerful. In language processing, they facilitate the discovery of related documents or sentences. By comparing the vector embeddings of different texts, finding similar or related information becomes faster and more efficient. This capability benefits search engines and recommendation systems, which can suggest relevant articles or products based on user interests.

In the realm of image analysis, vector databases excel in finding visually similar images. By representing images as vectors, a simple comparison of vector values can identify visually similar images. This capability is precious for tasks like reverse image search or content-based image retrieval.

Additionally, vector databases find applications in fraud detection, anomaly detection, and clustering. By comparing vector embeddings of data points, unusual patterns can be detected, and similar items can be grouped together, aiding in effective data analysis and decision-making.

Here is a list of the most popular vector databases:

- ChromaDB is a powerful database solution that stores and retrieves vector embeddings efficiently. It is commonly used in AI applications, including chatbots and document analysis systems. By storing embeddings in ChromaDB, users can easily search and retrieve similar vectors, enabling faster and more accurate matching or recommendation processes. ChromaDB offers excellent scalability high performance, and supports various indexing techniques to optimize search operations. It is a versatile tool that enhances the functionality and efficiency of AI applications that rely on vector embeddings.
- Facebook AI Similarity Search (FAISS) is another widely used vector database. Facebook AI Research develops it and offers highly optimized algorithms for similarity search and clustering of vector embeddings. FAISS is known for its speed and scalability, making it suitable for large-scale applications. It offers different indexing methods like flat, IVF (Inverted File System), and HNSW (Hierarchical Navigable Small World) to organize and search vector data efficiently.
- SingleStore: SingleStore aims to deliver the world's fastest distributed SQL database for data-intensive applications: SingleStoreDB, which combines transactional + analytical workloads in a single platform.

Skip to primary navigation
Skip to main content

- Astra DB: DataStax Astra DB is a cloud-native, multi-cloud, fully managed database-as-a-service based on Apache Cassandra, which aims to accelerate application development and reduce deployment time for applications from weeks to minutes.
- Milvus: Milvus is an open source vector database built to power embedding similarity search and AI applications. Milvus makes unstructured data search more accessible and provides a consistent user experience regardless of the deployment environment. Milvus 2.0 is a cloud-native vector database with storage and computation separated by design. All components in this refactored version of Milvus are stateless to enhance elasticity and flexibility.
- Qdrant: Qdrant is a vector similarity search engine and database for AI applications. Along with open-source, Qdrant is also available in the cloud. It provides a production-ready service with an API to store, search, and manage points—vectors with an additional payload. Qdrant is tailored to extended filtering support. It makes it useful for all sorts of neural network or semantic-based matching, faceted search, and other applications.
- Pinecone: Pinecone is a fully managed vector database that makes adding vector search to production applications accessible. It combines state-of-the-art vector search libraries, advanced features such as filtering, and distributed infrastructure to provide high performance and reliability at any scale.
- Vespa: Vespa is a platform for applications combining data and AI online. Building such applications on Vespa helps users avoid integration work to get features, and it can scale to support any amount of traffic and data. To deliver that, Vespa provides a broad range of query capabilities, a computation engine with support for modern machine-learned models, hands-off operability, data management, and application development support. It is free and open source to use under the Apache 2.0 license.
- Zilliz: Milvus is an open-source vector database, with over 18,409 stars on GitHub and 3.4 million+ downloads. Milvus supports billion-scale vector search and has over 1,000 enterprise users. Zilliz Cloud provides a fully-managed Milvus service made by the creators of Milvus. This helps to simplify the process of deploying and scaling vector search applications by eliminating the need to create and maintain complex data infrastructure. As a DBaaS, Zilliz simplifies the process of deploying and scaling vector search applications by eliminating the need to create and maintain complex data infrastructure.
- Weaviate: Weaviate is an open-source vector database used to store data objects and vector embeddings from ML-models, and scale into billions of data objects from the same name company in Amsterdam. Users can index billions of data objects to search through and combine multiple search techniques, such as keyword-based and vector search, to provide search experiences.

This sample makes of ChromaDB vector database, but you can easily modify the code to use another vector database. You can even use Azure Cache for Redis Enterprise to store the vector embeddings and compute vector similarity with high performance and low latency. For more information, see Vector Similarity Search with Azure Cache for Redis Enterprise

LangChain

LangChain is a software framework designed to streamline the development of applications using large language models (LLMs). It serves as a language model integration framework, facilitating various applications like document analysis and summarization, chatbots, and code analysis.

LangChain's integrations cover an extensive range of systems, tools, and services, making it a comprehensive solution for language model-based applications. LangChain integrates with the major cloud platforms such as Microsoft Azure, Amazon AWS, and Google, and with API wrappers for various purposes like news, movie information, and weather, as well as support for Bash, web scraping, and more. It also supports multiple language models, including those from OpenAI, Anthropic, and Hugging Face. Moreover, LangChain offers various functionalities for document handling, code generation, analysis, debugging, and interaction with databases and other data sources.

Chainlit

Chainlit is an open-source Python package that is specifically designed to create user interfaces (UIs) for AI applications. It simplifies the process of building interactive chats and interfaces, making developing AI-powered applications faster and more efficient. While Streamlit is a general-purpose UI library, Chainlit is purpose-built for AI applications and seamlessly integrates with other AI technologies such as LangChain, LlamaIndex, and LangFlow.

With Chainlit, developers can easily create intuitive UIs for their AI models, including ChatGPT-like applications. It provides a user-friendly interface for users to interact with AI models, enabling conversational experiences and information retrieval. Chainlit also offers unique features, such as displaying the Chain of Thought, which allows users to explore the reasoning process directly within the UI. This feature enhances transparency and enables users to understand how the AI arrives at its responses or recommendations.

For more information, see the following resources:

- Documentation
- Examples
- API Reference
- Cookbook

Deploy the Infrastructure

Before deploying the Terraform modules in the `terraform/infra` folder, specify a value for the following variables in the terraform.tfvars variable definitions file.

```
name_prefix = "Blue"
location    = "EastUS"
```

This is the definition of each variable:

- `prefix` : specifies a prefix for all the Azure resources.
- `location` : specifies the region (e.g., EastUS) where deploying the Azure resources.

NOTE: Make sure to select a region where Azure OpenAI Service (AOAI) supports both GPT-3.5/GPT-4 models like `gpt-35-turbo-16k` and Embeddings models like `text-embedding-ada-002` .

OpenAI Module

The following table contains the code from the `terraform/infra/modules/openai/main.tf` Terraform module used to deploy the Azure OpenAI Service.

```
resource "azurerm_cognitive_account" "openai" {
  name                          = var.name
  location                      = var.location
  resource_group_name           = var.resource_group_name
  kind                          = "OpenAI"
  custom_subdomain_name         = var.custom_subdomain_name
  sku_name                      = var.sku_name
  public_network_access_enabled = var.public_network_access_enabled
  tags                          = var.tags

  identity {
    type = "SystemAssigned"
  }

  lifecycle {
    ignore_changes = [
      tags
    ]
  }
}

resource "azurerm_cognitive_deployment" "deployment" {
  for_each             = {for deployment in var.deployments: deployment.name => deployment}

  name                 = each.key
  cognitive_account_id = azurerm_cognitive_account.openai.id

  model {
    format  = "OpenAI"
    name    = each.value.model.name
    version = each.value.model.version
  }

  scale {
    type = "Standard"
  }
}

resource "azurerm_monitor_diagnostic_setting" "settings" {
  name                       = "DiagnosticsSettings"
  target_resource_id         = azurerm_cognitive_account.openai.id
  log_analytics_workspace_id = var.log_analytics_workspace_id

  enabled_log {
    category = "Audit"

    retention_policy {
      enabled = true
      days    = var.log_analytics_retention_days
    }
  }

  enabled_log {
    category = "RequestResponse"

    retention_policy {
      enabled = true
      days    = var.log_analytics_retention_days
    }

```

Skip to primary navigation
Skip to main content

```
  }

  enabled_log {
    category = "Trace"

    retention_policy {
      enabled = true
      days    = var.log_analytics_retention_days
    }
  }

  metric {
    category = "AllMetrics"

    retention_policy {
      enabled = true
      days    = var.log_analytics_retention_days
    }
  }
}
```

Azure Cognitive Services uses custom subdomain names for each resource created through the Azure portal, Azure Cloud Shell, Azure CLI, Bicep, Azure Resource Manager (ARM), or Terraform. Unlike regional endpoints, which were common for all customers in a specific Azure region, custom subdomain names are unique to the resource. Custom subdomain names are required to enable authentication features like Azure Active Directory (Azure AD). We need to specify a custom subdomain for our Azure OpenAI Service, as our chatbot applications will use an Azure AD security token to access it. By default, the `terraform/infra/modules/openai/main.tf` module sets the value of the `custom_subdomain_name` parameter to the lowercase name of the Azure OpenAI resource. For more information on custom subdomains, see Custom subdomain names for Cognitive Services.

This Terraform module allows you to pass an array containing the definition of one or more model deployments in the `deployments` variable. For more information on model deployments, see Create a resource and deploy a model using Azure OpenAI. The `openai_deployments` variable in the `terraform/infra/variables.tf` file defines the structure and the default models deployed by the sample:

```
variable "openai_deployments" {
  description = "(Optional) Specifies the deployments of the Azure OpenAI Service"
  type = list(object({
    name = string
    model = object({
      name = string
      version = string
    })
    rai_policy_name = string
  }))
  default = [
    {
      name = "gpt-35-turbo-16k"
      model = {
        name = "gpt-35-turbo-16k"
        version = "0613"
      }
      rai_policy_name = ""
    },
    {
      name = "text-embedding-ada-002"
      model = {
        name = "text-embedding-ada-002"
        version = "2"
      }
      rai_policy_name = ""
    }
  ]
}
```

Alternatively, you can use the Terraform module for deploying Azure OpenAI Service. to deploy Azure OpenAI Service.

Private Endpoint Module

The `terraform/infra/main.tf` the module creates Azure Private Endpoints and Azure Private DNDS Zones for each of the following resources:

- Azure OpenAI Service (AOAI)
- Azure Container Registry (ACR)

In particular, it creates an Azure Private Endpoint and Azure Private DNDS Zone to the Azure OpenAI Service as shown in the following code snippet:

Skip to main content
Skip to primary navigation

```
module "openai_private_dns_zone" {
  source                       = "./modules/private_dns_zone"
  name                         = "privatelink.openai.azure.com"
  resource_group_name          = azurerm_resource_group.rg.name
  tags                         = var.tags
  virtual_networks_to_link     = {
    (module.virtual_network.name) = {
      subscription_id = data.azurerm_client_config.current.subscription_id
      resource_group_name = azurerm_resource_group.rg.name
    }
  }
}

module "openai_private_endpoint" {
  source                         = "./modules/private_endpoint"
  name                           = "${module.openai.name}PrivateEndpoint"
  location                       = var.location
  resource_group_name            = azurerm_resource_group.rg.name
  subnet_id                      = module.virtual_network.subnet_ids[var.vm_subnet_name]
  tags                           = var.tags
  private_connection_resource_id = module.openai.id
  is_manual_connection           = false
  subresource_name               = "account"
  private_dns_zone_group_name    = "AcrPrivateDnsZoneGroup"
  private_dns_zone_group_ids     = [module.openai_private_dns_zone.id]
}
```

Below you can read the code of the `terraform/infra/modules/private_endpoint/main.tf` module, which is used to create Azure Private Endpoints:

Skip to primary navigation
Skip to main content

```
resource "azurerm_private_endpoint" "private_endpoint" {
  name                = var.name
  location            = var.location
  resource_group_name = var.resource_group_name
  subnet_id           = var.subnet_id
  tags                = var.tags

  private_service_connection {
    name                           = "${var.name}Connection"
    private_connection_resource_id = var.private_connection_resource_id
    is_manual_connection           = var.is_manual_connection
    subresource_names              = try([var.subresource_name], null)
    request_message                = try(var.request_message, null)
  }

  private_dns_zone_group {
    name                 = var.private_dns_zone_group_name
    private_dns_zone_ids = var.private_dns_zone_group_ids
  }

  lifecycle {
    ignore_changes = [
      tags
    ]
  }
}
```

Private DNS Zone Module

In the following box, you can read the code of the `terraform/infra/modules/private_dns_zone/main.tf` module, which is utilized to create the Azure Private DNS Zones.

```
resource "azurerm_private_dns_zone" "private_dns_zone" {
  name                = var.name
  resource_group_name = var.resource_group_name
  tags                = var.tags

  lifecycle {
    ignore_changes = [
      tags
    ]
  }
}

resource "azurerm_private_dns_zone_virtual_network_link" "link" {
  for_each = var.virtual_networks_to_link

  name                  = "link_to_${lower(basename(each.key))}"
  resource_group_name   = var.resource_group_name
  private_dns_zone_name = azurerm_private_dns_zone.private_dns_zone.name
  virtual_network_id    = "/subscriptions/${each.value.subscription_id}/resourceGroups/${each.val

  lifecycle {
    ignore_changes = [
      tags
    ]
  }
}
```

Workload Managed Identity Module

Below you can read the code of the `terraform/infra/modules/managed_identity/main.tf` module, which is used to create the Azure Managed Identity used by the Azure Container Apps to pull container images from the Azure Container Registry, and by the chat applications to connect to the Azure OpenAI Service. You can use a system-assigned or user-assigned managed identity from Azure Active Directory (Azure AD) to let Azure Container Apps access any Azure AD-protected resource. For more information, see Managed identities in Azure Container Apps. You can pull container images from private repositories in an Azure Container Registry using user-assigned or user-assigned managed identities for authentication to avoid using administrative credentials. For more information, see Azure Container Apps image pull with managed identity. This user-defined managed identity is assigned the Cognitive Services User role on the Azure OpenAI Service namespace and ACRPull role on the Azure Container Registry (ACR). By assigning the above roles, you grant the user-defined managed identity access to these resources.

```
resource "azurerm_user_assigned_identity" "workload_user_assigned_identity" {
  name                = var.name
  resource_group_name = var.resource_group_name
  location            = var.location
  tags                = var.tags

  lifecycle {
    ignore_changes = [
      tags
    ]
  }
}

resource "azurerm_role_assignment" "cognitive_services_user_assignment" {
  scope                = var.openai_id
  role_definition_name = "Cognitive Services User"
  principal_id         = azurerm_user_assigned_identity.workload_user_assigned_identity.principal_
  skip_service_principal_aad_check = true
}

resource "azurerm_role_assignment" "acr_pull_assignment" {
  scope                = var.acr_id
  role_definition_name = "AcrPull"
  principal_id         = azurerm_user_assigned_identity.workload_user_assigned_identity.principal_
  skip_service_principal_aad_check = true
}
```

Deploy the Applications

Before deploying the Terraform modules in the `terraform/apps` folder, specify a value for the following variables in the Terraform.tfvars variable definitions file.

```
resource_group_name            = "BlueRG"
container_app_environment_name = "BlueEnvironment"
container_registry_name        = "BlueRegistry"
workload_managed_identity_name = "BlueWorkloadIdentity"
container_apps                 = [
  {
    name                           = "chatapp"
    revision_mode                  = "Single"
    ingress                        = {
      allow_insecure_connections   = true
      external_enabled             = true
      target_port                  = 8000
      transport                    = "http"
      traffic_weight               = {
        label                      = "default"
        latest_revision            = true
        revision_suffix            = "default"
        percentage                 = 100
      }
    }
    template                       = {
      containers                   = [
        {
          name                     = "chat"
          image                    = "chat:v1"
          cpu                      = 0.5
          memory                   = "1Gi"
          env                      = [
            {
              name                 = "TEMPERATURE"
              value                = 0.9
            },
            {
              name                 = "AZURE_OPENAI_BASE"
              value                = "https://blueopenai.openai.azure.com/"
            },
            {
              name                 = "AZURE_OPENAI_KEY"
              value                = ""
            },
            {
              name                 = "AZURE_OPENAI_TYPE"
              value                = "azure_ad"
            },
            {
              name                 = "AZURE_OPENAI_VERSION"
              value                = "2023-06-01-preview"
            },
            {
              name                 = "AZURE_OPENAI_DEPLOYMENT"
              value                = "gpt-35-turbo-16k"
            },
            {
              name                 = "AZURE_OPENAI_MODEL"
              value                = "gpt-35-turbo-16k"
            },
            {
              name                 = "AZURE_OPENAI_SYSTEM_MESSAGE"
              value                = "You are a helpful assistant."

```

Skip to main content

```
            },
            {
              name                  = "MAX_RETRIES"
              value                 = 5
            },
            {
              name                  = "BACKOFF_IN_SECONDS"
              value                 = "1"
            },
            {
              name                  = "TOKEN_REFRESH_INTERVAL"
              value                 = 2700
            }
          ]
          liveness_probe            = {
            failure_count_threshold = 3
            initial_delay           = 30
            interval_seconds        = 60
            path                    = "/"
            port                    = 8000
            timeout                 = 30
            transport               = "HTTP"
          }
          readiness_probe = {
            failure_count_threshold = 3
            interval_seconds        = 60
            path                    = "/"
            port                    = 8000
            success_count_threshold = 3
            timeout                 = 30
            transport               = "HTTP"
          }
          startup_probe = {
            failure_count_threshold = 3
            interval_seconds        = 60
            path                    = "/"
            port                    = 8000
            timeout                 = 30
            transport               = "HTTP"
          }
        }
      ]
      min_replicas                  = 1
      max_replicas                  = 3
    }
  },
  {
    name                            = "docapp"
    revision_mode                   = "Single"
    ingress                         = {
      allow_insecure_connections    = true
      external_enabled              = true
      target_port                   = 8000
      transport                     = "http"
      traffic_weight                = {
        label                       = "default"
        latest_revision             = true
        revision_suffix             = "default"
        percentage                  = 100
      }
```

Skip to main content Skip to navigation

```
    }
    template                        = {
      containers                    = [
        {
          name                      = "doc"
          image                     = "doc:v1"
          cpu                       = 0.5
          memory                    = "1Gi"
          env                       = [
            {
              name                  = "TEMPERATURE"
              value                 = 0.9
            },
            {
              name                  = "AZURE_OPENAI_BASE"
              value                 = "https://blueopenai.openai.azure.com/"
            },
            {
              name                  = "AZURE_OPENAI_KEY"
              value                 = ""
            },
            {
              name                  = "AZURE_OPENAI_TYPE"
              value                 = "azure_ad"
            },
            {
              name                  = "AZURE_OPENAI_VERSION"
              value                 = "2023-06-01-preview"
            },
            {
              name                  = "AZURE_OPENAI_DEPLOYMENT"
              value                 = "gpt-35-turbo-16k"
            },
            {
              name                  = "AZURE_OPENAI_MODEL"
              value                 = "gpt-35-turbo-16k"
            },
            {
              name                  = "AZURE_OPENAI_ADA_DEPLOYMENT"
              value                 = "text-embedding-ada-002"
            },
            {
              name                  = "AZURE_OPENAI_SYSTEM_MESSAGE"
              value                 = "You are a helpful assistant."
            },
            {
              name                  = "MAX_RETRIES"
              value                 = 5
            },
            {
              name                  = "CHAINLIT_MAX_FILES"
              value                 = 10
            },
            {
              name                  = "TEXT_SPLITTER_CHUNK_SIZE"
              value                 = 1000
            },
            {
              name                  = "TEXT_SPLITTER_CHUNK_OVERLAP"
              value                 = 100
```

Skip to primary navigation
Skip to main content

```
            },
            {
              name                  = "EMBEDDINGS_CHUNK_SIZE"
              value                 = 16
            },
            {
              name                  = "BACKOFF_IN_SECONDS"
              value                 = "1"
            },
            {
              name                  = "CHAINLIT_MAX_SIZE_MB"
              value                 = 100
            },
            {
              name                  = "TOKEN_REFRESH_INTERVAL"
              value                 = 2700
            }
          ]
          liveness_probe = {
            failure_count_threshold = 3
            initial_delay           = 30
            interval_seconds        = 60
            path                    = "/"
            port                    = 8000
            timeout                 = 30
            transport               = "HTTP"
          }
          readiness_probe = {
            failure_count_threshold = 3
            interval_seconds        = 60
            path                    = "/"
            port                    = 8000
            success_count_threshold = 3
            timeout                 = 30
            transport               = "HTTP"
          }
          startup_probe = {
            failure_count_threshold = 3
            interval_seconds        = 60
            path                    = "/"
            port                    = 8000
            timeout                 = 30
            transport               = "HTTP"
          }
        }
      ]
      min_replicas                  = 1
      max_replicas                  = 3
    }
  }]
```

This is the definition of each variable:

- `resource_group_name` : specifies the name of the resource group that contains the infrastructure resources: Azure OpenAI Service, Azure Container Registry, Azure Container Apps Environment, Azure Log Analytics, and user-defined managed identity.
- `container_app_environment_name` : the name of the Azure Container Apps Environment in which to deploy the chat applications.

Skip to main content Skip to navigation

- `container_registry_name` : the name of Azure Container Registry used to hold the container images of the chat applications.
- `workload_managed_identity_name` : the name of the user-defined managed identity used by the chat applications to authenticate with Azure OpenAI Service and Azure Container Registry.
- `container_apps` : the definition of the two chat applications. The application configuration does not specify the following data because the `container_app` module later defines this information:
    - `image` : This field contains the name and tag of the container image but not the login server of the Azure Container Registry.
    - `identity` : The identity of the container app.
    - `registry` : The registry hosting the container image for the application.
    - `AZURE_CLIENT_ID` : The client ID of the user-defined managed identity used by the application to authenticate with Azure OpenAI Service and Azure Container Registry.
    - `AZURE_OPENAI_TYPE` : This environment variable specifies the authentication type with Azure OpenAI Service: if you set the value of the `AZURE_OPENAI_TYPE` environment variable to `azure` , you need to specify the OpenAI key as a value for the `AZURE_OPENAI_KEY` environment variable. Instead, if you set the value to `azure_ad` in the application code, assign an Azure AD security token to the `openai_api_key` property. For more information, see How to switch between OpenAI and Azure OpenAI endpoints with Python.

Container App Module

The `terraform/apps/modules/container_app/main.tf` module is utilized to create the Azure Container Apps. The module defines and uses the following data source for the Azure Container Registry, Azure Container Apps Environment, and user-defined managed identity created when deploying the infrastructure. These data sources are used to access the properties of these Azure resources.

```
data "azurerm_container_app_environment" "container_app_environment" {
  name                 = var.container_app_environment_name
  resource_group_name  = var.resource_group_name
}

data "azurerm_container_registry" "container_registry" {
  name                 = var.container_registry_name
  resource_group_name  = var.resource_group_name
}

data "azurerm_user_assigned_identity" "workload_user_assigned_identity" {
  name                = var.workload_managed_identity_name
  resource_group_name = var.resource_group_name
}
```

The module creates and utilizes the following local variables:

Skip to main content Skip to primary navigation

```
locals {
  identity = {
    type         = "UserAssigned"
    identity_ids = [data.azurerm_user_assigned_identity.workload_user_assigned_identity.id]
  }
  identity_env = {
    name         = "AZURE_CLIENT_ID"
    secret_name  = null
    value        = data.azurerm_user_assigned_identity.workload_user_assigned_identity.client_id
  }
  registry = {
    server       = data.azurerm_container_registry.container_registry.login_server
    identity     = data.azurerm_user_assigned_identity.workload_user_assigned_identity.id
  }
}
```

This is the explanation of each local variable:

- `identity` : uses the resource ID of the user-defined managed identity to define the `identity` block for each container app deployed by the module.
- `identity_env` : uses the client ID of the user-defined managed identity to define the value of the `AZURE_CLIENT_ID` environment variable that is appended to the list of environment variables of each container app deployed by the module.
- `registry` : uses the login server of the Azure Container Registry to define the `registry` block for each container app deployed by the module.

Here is the complete Terraform code of the module:

```
data "azurerm_container_app_environment" "container_app_environment" {
  name                 = var.container_app_environment_name
  resource_group_name  = var.resource_group_name
}

data "azurerm_container_registry" "container_registry" {
  name                 = var.container_registry_name
  resource_group_name  = var.resource_group_name
}

data "azurerm_user_assigned_identity" "workload_user_assigned_identity" {
  name                = var.workload_managed_identity_name
  resource_group_name = var.resource_group_name
}

locals {
  identity = {
    type         = "UserAssigned"
    identity_ids = [data.azurerm_user_assigned_identity.workload_user_assigned_identity.id]
  }
  identity_env = {
    name         = "AZURE_CLIENT_ID"
    secret_name  = null
    value        = data.azurerm_user_assigned_identity.workload_user_assigned_identity.client_id
  }
  registry = {
    server       = data.azurerm_container_registry.container_registry.login_server
    identity     = data.azurerm_user_assigned_identity.workload_user_assigned_identity.id
  }
}

resource "azurerm_container_app" "container_app" {
  for_each                     = {for app in var.container_apps: app.name => app}

  container_app_environment_id = data.azurerm_container_app_environment.container_app_environment
  name                         = each.key
  resource_group_name          = var.resource_group_name
  revision_mode                = each.value.revision_mode
  tags                         = each.value.tags

  template {
    max_replicas    = each.value.template.max_replicas
    min_replicas    = each.value.template.min_replicas
    revision_suffix = each.value.template.revision_suffix

    dynamic "container" {
      for_each = each.value.template.containers

      content {
        cpu     = container.value.cpu
        image   = "${data.azurerm_container_registry.container_registry.login_server}/${container
        memory  = container.value.memory
        name    = container.value.name
        args    = container.value.args
        command = container.value.command

        dynamic "env" {
          for_each = container.value.env == null ? [local.identity_env] : concat(container.value.
```

Skip to primary navigation
Skip to main content

```
  content {
    name        = env.value.name
    secret_name = env.value.secret_name
    value       = env.value.value
  }
}

dynamic "liveness_probe" {
  for_each = container.value.liveness_probe == null ? [] : [container.value.liveness_prob

  content {
    port                    = liveness_probe.value.port
    transport               = liveness_probe.value.transport
    failure_count_threshold = liveness_probe.value.failure_count_threshold
    host                    = liveness_probe.value.host
    initial_delay           = liveness_probe.value.initial_delay
    interval_seconds        = liveness_probe.value.interval_seconds
    path                    = liveness_probe.value.path
    timeout                 = liveness_probe.value.timeout

    dynamic "header" {
      for_each = liveness_probe.value.header == null ? [] : [liveness_probe.value.header]

      content {
        name  = header.value.name
        value = header.value.value
      }
    }
  }
}

dynamic "readiness_probe" {
  for_each = container.value.readiness_probe == null ? [] : [container.value.readiness_pr

  content {
    port                    = readiness_probe.value.port
    transport               = readiness_probe.value.transport
    failure_count_threshold = readiness_probe.value.failure_count_threshold
    host                    = readiness_probe.value.host
    interval_seconds        = readiness_probe.value.interval_seconds
    path                    = readiness_probe.value.path
    success_count_threshold = readiness_probe.value.success_count_threshold
    timeout                 = readiness_probe.value.timeout

    dynamic "header" {
      for_each = readiness_probe.value.header == null ? [] : [readiness_probe.value.heade

      content {
        name  = header.value.name
        value = header.value.value
      }
    }
  }
}

dynamic "startup_probe" {
  for_each = container.value.startup_probe == null ? [] : [container.value.startup_probe]

  content {
    port                                            .port
```

```
          transport               = startup_probe.value.transport
          failure_count_threshold = startup_probe.value.failure_count_threshold
          host                    = startup_probe.value.host
          interval_seconds        = startup_probe.value.interval_seconds
          path                    = startup_probe.value.path
          timeout                 = startup_probe.value.timeout

          dynamic "header" {
            for_each = startup_probe.value.header == null ? [] : [startup_probe.value.header]

            content {
              name  = header.value.name
              value = header.value.name
            }
          }
        }
      }

      dynamic "volume_mounts" {
        for_each = container.value.volume_mounts == null ? [] : [container.value.volume_mounts]

        content {
          name = volume_mounts.value.name
          path = volume_mounts.value.path
        }
      }
    }
  }

  dynamic "volume" {
    for_each = each.value.template.volume == null ? [] : each.value.template.volume

    content {
      name         = volume.value.name
      storage_name = volume.value.storage_name
      storage_type = volume.value.storage_type
    }
  }
}

dynamic "dapr" {
  for_each = each.value.dapr == null ? [] : [each.value.dapr]

  content {
    app_id       = dapr.value.app_id
    app_port     = dapr.value.app_port
    app_protocol = dapr.value.app_protocol
  }
}

dynamic "identity" {
  for_each = each.value.identity == null ? [local.identity] : [each.value.identity]

  content {
    type         = identity.value.type
    identity_ids = identity.value.identity_ids
  }
}

dynamic "ingress" {
```

Skip to primary navigation Skip to main content

```
    for_each = each.value.ingress == null ? [] : [each.value.ingress]

    content {
      target_port                = ingress.value.target_port
      allow_insecure_connections = ingress.value.allow_insecure_connections
      external_enabled           = ingress.value.external_enabled
      transport                  = ingress.value.transport

      dynamic "traffic_weight" {
        for_each = ingress.value.traffic_weight == null ? [] : [ingress.value.traffic_weight]

        content {
          percentage      = traffic_weight.value.percentage
          label           = traffic_weight.value.label
          latest_revision = traffic_weight.value.latest_revision
          revision_suffix = traffic_weight.value.revision_suffix
        }
      }
    }
  }

  dynamic "registry" {
    for_each = each.value.registry == null ? [local.registry] : concat(each.value.registry, [loca

    content {
      server   = registry.value.server
      identity = registry.value.identity
    }
  }

  dynamic "secret" {
    for_each = nonsensitive(toset([for pair in lookup(var.container_app_secrets, each.key, []) :

    content {
      name  = secret.key
      value = local.container_app_secrets[each.key][secret.key]
    }
  }
}
```

As you can notice, the module uses the login server of the Azure Container Registry to create the fully qualified name of the container image of the current container app.

Managed identities in Azure Container Apps

Each chat application makes use of a DefaultAzureCredential object to acquire a security token from Azure Active Directory and authenticate and authorize with Azure OpenAI Service (AOAI) and Azure Container Registry (ACR) using the credentials of the user-defined managed identity associated with the container app. You can use a managed identity in a running container app to authenticate and authorize with any service that supports Azure AD authentication. With managed identities:

- Container apps and applications connect to resources with the managed identity. You don't need to manage credentials in your container apps.
- You can use role-based access control to grant specific permissions to a managed identity.

- System-assigned identities are automatically created and managed. They are deleted when your container app or container app is deleted.
- You can add and delete user-assigned identities and assign them to multiple resources. They are independent of your container app or the container app's lifecycle.
- You can use managed identity to authenticate with a private Azure Container Registry without a username and password to pull containers for your Container App.
- You can use managed identity to create connections for Dapr-enabled applications via Dapr components

For more information, see Managed identities in Azure Container Apps. The workloads running in a container app can use the Azure Identity client libraries to acquire a security token from the Azure Active Directory. You can choose one of the following approaches inside your code:

- Use `DefaultAzureCredential`, which will attempt to use the `WorkloadIdentityCredential`.
- Create a `ChainedTokenCredential` instance that includes `WorkloadIdentityCredential`.
- Use `WorkloadIdentityCredential` directly.

The following table provides the minimum package version required for each language's client library.

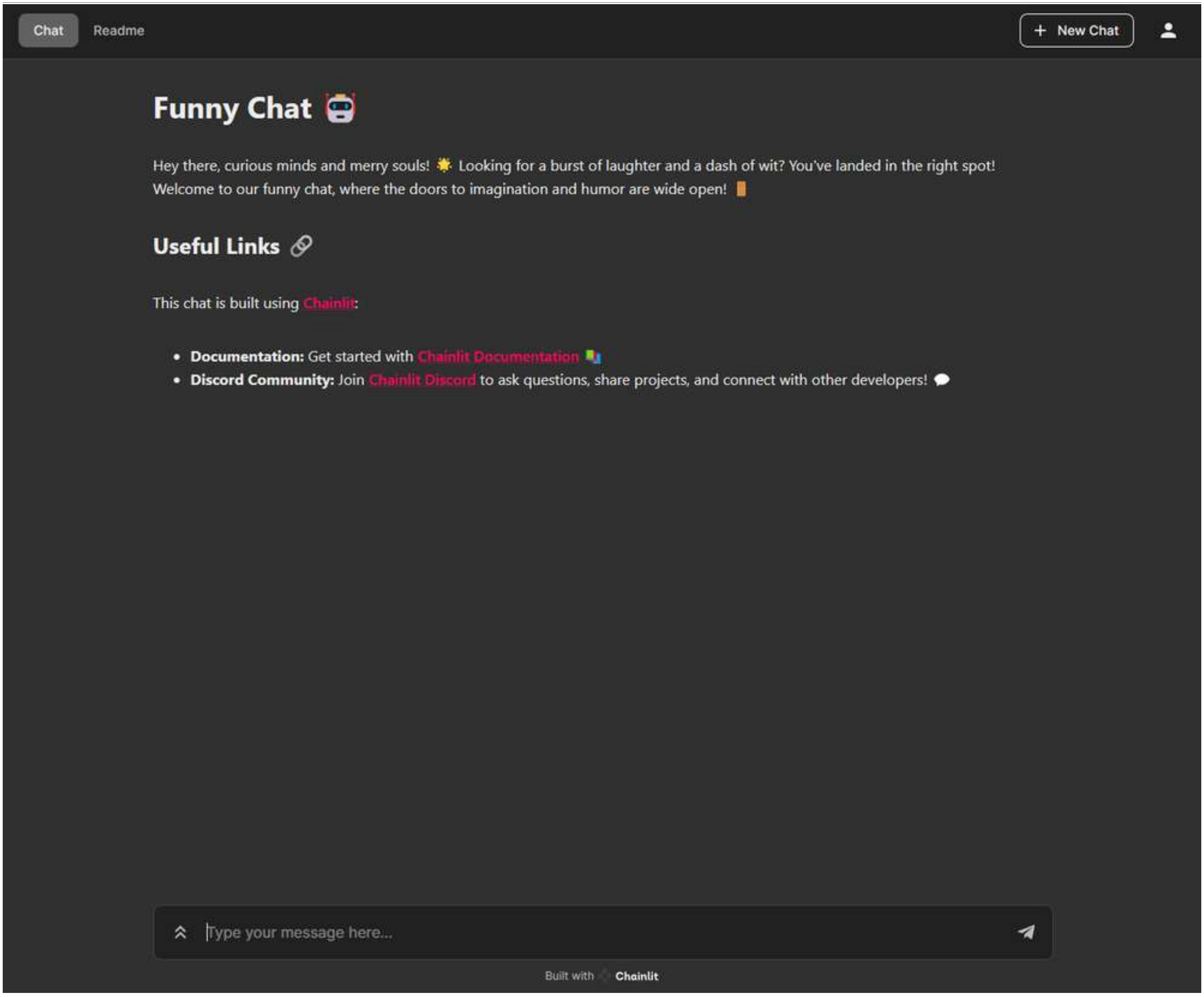
Language	Library	Minimum Version	Example
.NET	Azure.Identity	1.9.0	Link
Go	azidentity	1.3.0	Link
Java	azure-identity	1.9.0	Link
JavaScript	@azure/identity	3.2.0	Link
Python	azure-identity	1.13.0	Link

NOTE: When using Azure Identity client library with Azure Container Apps, the client ID of the managed identity must be specified. When using the `DefaultAzureCredential`, you can explicitly specify the client ID of the container app managed identity in the `AZURE_CLIENT_ID` environment variable.

Simple Chat Application

The Simple Chat Application is a large language model-based chatbot that allows users to submit general-purpose questions to a GPT model, generating and streaming back human-like and engaging conversational responses. The following picture shows the welcome screen of the chat application.

Skip to primary navigation
Skip to main content



Funny Chat 🤖

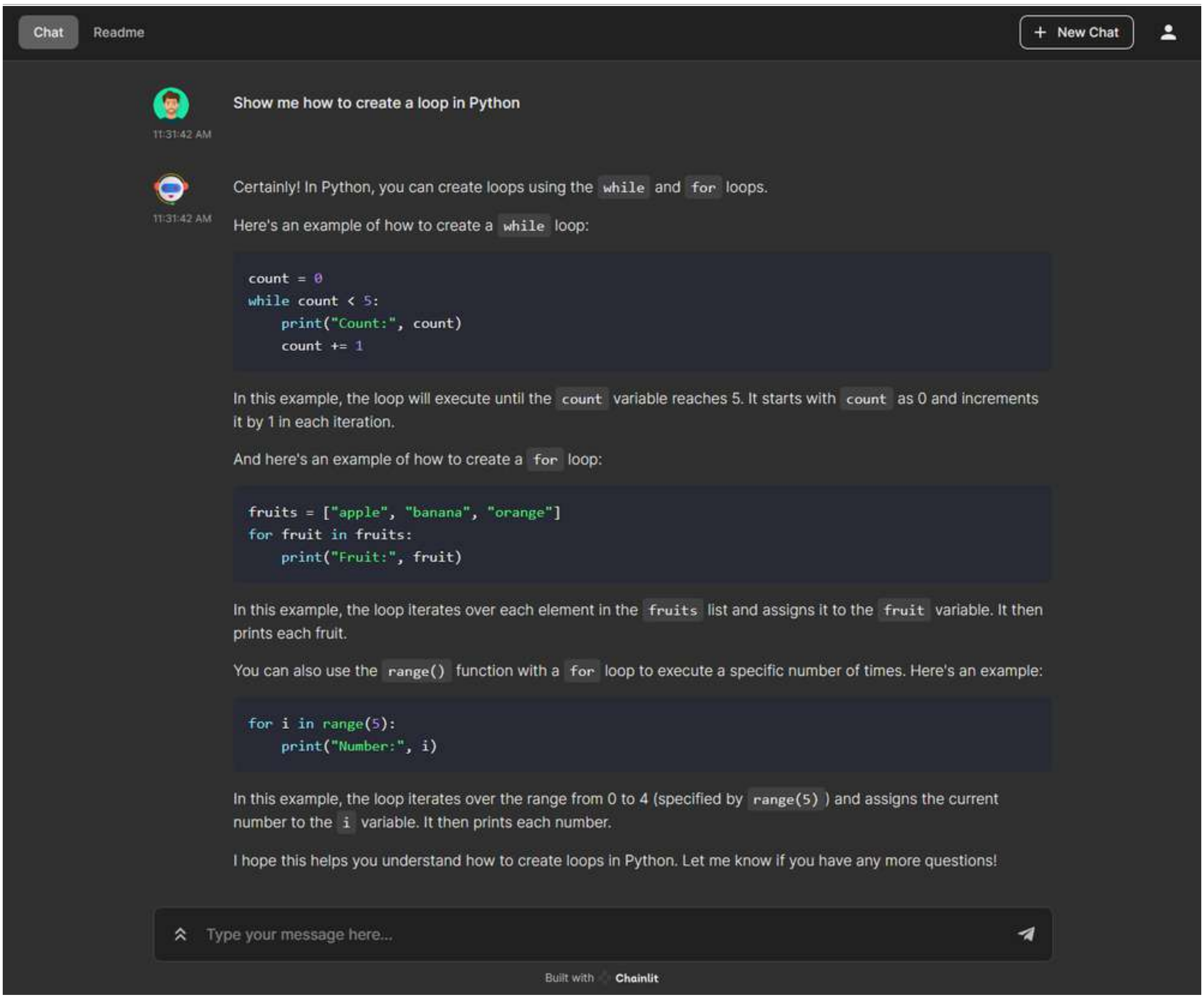
Hey there, curious minds and merry souls! 🌟 Looking for a burst of laughter and a dash of wit? You've landed in the right spot! Welcome to our funny chat, where the doors to imagination and humor are wide open! 🚪

Useful Links 🔗

This chat is built using Chainlit:

- **Documentation:** Get started with Chainlit Documentation 📚
- **Discord Community:** Join Chainlit Discord to ask questions, share projects, and connect with other developers! 💬

You can modify the welcome screen in markdown by editing the `chainlit.md` file at the project's root. If you do not want a welcome screen, leave the file empty. The following picture shows what happens when a user submits a new message in the chat.



Show me how to create a loop in Python

11:31:42 AM

Certainly! In Python, you can create loops using the `while` and `for` loops.

11:31:42 AM

Here's an example of how to create a `while` loop:

```
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

In this example, the loop will execute until the `count` variable reaches 5. It starts with `count` as 0 and increments it by 1 in each iteration.

And here's an example of how to create a `for` loop:

```
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print("Fruit:", fruit)
```

In this example, the loop iterates over each element in the `fruits` list and assigns it to the `fruit` variable. It then prints each fruit.

You can also use the `range()` function with a `for` loop to execute a specific number of times. Here's an example:

```
for i in range(5):
    print("Number:", i)
```

In this example, the loop iterates over the range from 0 to 4 (specified by `range(5)`) and assigns the current number to the `i` variable. It then prints each number.

I hope this helps you understand how to create loops in Python. Let me know if you have any more questions!

Chainlit can render messages in markdown format and provides classes to support the following elements:

- Audio: The `Audio` class allows you to display an audio player for a specific audio file in the chatbot user interface. You must provide either a URL or a path or content bytes.
- Avatar: The `Avatar` class allows you to display an avatar image next to a message instead of the author's name. You need to send the element once. Next,, if an avatar's name matches an author's name, the avatar will be automatically displayed. You must provide either a URL or a path or content bytes.
- File: The `File` class allows you to display a button that lets users download the file's content. You must provide either a URL or a path or content bytes.
- Image: The `Image` class is designed to create and handle image elements to be sent and displayed in the chatbot user interface. You must provide either a URL or a path or content bytes.
- Pdf: The `Pdf` class allows you to display a PDF hosted remotely or locally in the chatbot UI. This class either takes a URL of a PDF hosted online or the path of a local PDF.
- Pyplot: The `Pyplot` class allows you to display a Matplotlib pyplot chart in the chatbot UI. This class takes a pyplot figure.
- TaskList: The `TaskList` class allows you to display a task list next to the chatbot UI.
- Text: The `Text` class allows you to display a text element in the chatbot UI. This class takes a string and creates a text element that can be sent to the UI. It supports the markdown syntax for formatting text. You must provide either a URL or a path or content bytes.

Skip to primary navigation Skip to main content

Chainlit provides three display options that determine how an element is rendered in the context of its use. The ElementDisplay type represents these options. The following display options are available:

- `Side` : this option displays the element on a sidebar. The sidebar is hidden by default and opened upon element reference click.
- `Page` : this option displays the element on a separate page. The user is redirected to the page upon an element reference click.
- `Inline` : this option displays the element below the message. If the element is global, it is displayed if it is explicitly mentioned in the message. If the element is scoped, it is displayed regardless of whether it is expressly mentioned in the message.

You can click the user icon on the UI to access the chat settings and choose, for example, between the light and dark themes.
The application is built in Python. Let's take a look at the individual parts of the application code. The Python code starts by importing the necessary packages/modules in the following section.

```python
# Import packages
import os
import sys
import time
import openai
import random
import logging
import chainlit as cl
from azure.identity import DefaultAzureCredential
from dotenv import load_dotenv
from dotenv import dotenv_values

# Load environment variables from .env file
if os.path.exists(".env"):
    load_dotenv(override=True)
    config = dotenv_values(".env")
```

These are the libraries used by the chat application:

1. `os` : This module provides a way of interacting with the operating system, enabling the code to access environment variables, file paths, etc.
2. `sys` : This module provides access to some variables used or maintained by the interpreter and functions that interact with the interpreter.
3. `time` : This module provides various time-related time manipulation and measurement functions.
4. `openai` : The OpenAI Python library provides convenient access to the OpenAI API from applications written in Python. It includes a pre-defined set of classes for API resources that initialize themselves dynamically from API responses, making it compatible with a wide range of versions of the OpenAI API. You can find usage examples for the OpenAI Python library in our API reference and the OpenAI Cookbook.
5. `random` : This module provides functions to generate random numbers.
6. `logging` : This module provides flexible logging of messages.
7. `chainlit as cl` : This imports the Chainlit library and aliases it as `cl` . Chainlit is used to create the UI of the application.

Skip to main content
Skip to primary navigation

8. `DefaultAzureCredential` from `azure.identity` : when the `openai_type` property value is `azure_ad,` a `DefaultAzureCredential` object from the [Azure Identity client library for Python - version 1.13.0(https://learn.microsoft.com/en-us/python/api/overview/azure/identity-readme?view=azure-python) is used to acquire security token from the Azure Active Directory using the credentials of the user-defined managed identity, whose client ID is defined in the `AZURE_CLIENT_ID` environment variable.
9. `load_dotenv` and `dotenv_values` from `dotenv` : Python-dotenv reads key-value pairs from a `.env` file and can set them as environment variables. It helps in the development of applications following the 12-factor principles.

The `requirements.txt` file under the `src` folder contains the list of packages used by the chat applications. You can restore these packages in your environment using the following command:

```
pip install -r requirements.txt --upgrade
```

Next, the code reads environment variables and configures the OpenAI settings.

```
# Read environment variables
temperature = float(os.environ.get("TEMPERATURE", 0.9))
api_base = os.getenv("AZURE_OPENAI_BASE")
api_key = os.getenv("AZURE_OPENAI_KEY")
api_type = os.environ.get("AZURE_OPENAI_TYPE", "azure")
api_version = os.environ.get("AZURE_OPENAI_VERSION", "2023-06-01-preview")
engine = os.getenv("AZURE_OPENAI_DEPLOYMENT")
model = os.getenv("AZURE_OPENAI_MODEL")
system_content = os.getenv("AZURE_OPENAI_SYSTEM_MESSAGE", "You are a helpful assistant.")
max_retries = int(os.getenv("MAX_RETRIES", 5))
backoff_in_seconds = float(os.getenv("BACKOFF_IN_SECONDS", 1))

# Configure OpenAI
openai.api_type = api_type
openai.api_version = api_version
openai.api_base = api_base
openai.api_key = api_key
```

Here's a brief explanation of each variable and related environment variable:

1. `temperature` : A float value representing the temperature for Create chat completion method of the OpenAI API. It is fetched from the environment variables with a default value of 0.9.
2. `api_base` : The base URL for the OpenAI API.
3. `api_key` : The API key for the OpenAI API.
4. `api_type` : A string representing the type of the OpenAI API.
5. `api_version` : A string representing the version of the OpenAI API.
6. `engine` : The engine used for OpenAI API calls.
7. `model` : The model used for OpenAI API calls.
8. `system_content` : The content of the system message used for OpenAI API calls.
9. `max_retries` : The maximum number of retries for OpenAI API calls.
10. `backoff_in_seconds` : The backoff time in seconds for retries in case of failures.

In the next section, the code sets the default Azure credential based on the `api_type` and configures a logger for logging purposes.

```
# Set default Azure credential
default_credential = DefaultAzureCredential() if openai.api_type == "azure_ad" else None

# Configure a logger
logging.basicConfig(
    stream=sys.stdout,
    format='[%(asctime)s] {%(filename)s:%(lineno)d} %(levelname)s - %(message)s',
    level=logging.INFO
)
logger = logging.getLogger(__name__)
```

Here's a brief explanation:

1. `default_credential` : It sets the default Azure credential to `DefaultAzureCredential()` if the `api_type` is "azure_ad"; otherwise, it is set to `None` .
2. `logging.basicConfig()` : This function configures the logging system with specific settings.
    - `stream` : The output stream where log messages will be written. Here, it is set to `sys.stdout` for writing log messages to the standard output.
    - `format` : The format string for log messages. It includes the timestamp, filename, line number, log level, and the actual log message.
    - `level` : The logging level. It is set to `logging.INFO` , meaning only messages with the level `INFO` and above will be logged.
3. `logger` : This creates a logger instance named after the current module (`__name__`). The logger will be used to log messages throughout the code.

Next, the code defines a helper function `backoff` that takes an integer `attempt` and returns a float value representing the backoff time for exponential retries in case of API call failures.

```
def backoff(attempt: int) -> float:
    return backoff_in_seconds * 2 ** attempt + random.uniform(0, 1)
```

The backoff time is calculated using the `backoff_in_seconds` and `attempt` variables. It follows the formula `backoff_in_seconds * 2 ** attempt + random.uniform(0, 1)` . This formula increases the backoff time exponentially with each attempt and adds a random value between 0 and 1 to avoid synchronized retries. Then, the application defines a function called `refresh_openai_token()` to refresh the OpenAI security token if needed.

```
def refresh_openai_token():
    token = cl.user_session.get('openai_token')
    if token is None or token.expires_on < int(time.time()) - 1800:
        cl.user_session.set('openai_token', default_credential.get_token("https://cognitiveservic
        openai.api_key = cl.user_session.get('openai_token').token
```

The function follows these steps:

1. It fetches the current token from `cl.user_session` (which seems to be a part of the `chainlit` library) using the key `'openai_token'` . The user_session is a dictionary that stores the user's session data. The id and env keys are reserved for the session ID and environment variables. Other keys can be used to store arbitrary data in the user's session.

Skip to primary navigation
Skip to main content

2. It checks if the token is `None` or if its expiration time (`expires_on`) is less than the current time minus 1800 seconds (30 minutes).

Next, the code defines a function called `start_chat` that is used to initialize the when the user connects to the application or clicks the `New Chat` button.

```
.on_chat_start
async def start_chat():
    # Sending Avatars for Chat Participants
    await cl.Avatar(
        name="Chatbot",
        url="https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name="Error",
        url="https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name="User",
        url="https://media.architecturaldigest.com/photos/5f241de2c850b2a36b415024/master/w_1600%
    ).send()

    # Initializing message_history in user session
    system_content = "Welcome to the chat!"
    cl.user_session.set("message_history", [{"role": "system", "content": system_content}])
```

Here is a brief explanation of the function steps:

- `@cl.on_chat_start` : The on_chat_start decorator registers a callback function `start_chat()` to be called when the Chainlit chat starts. It is used to set up the chat and send avatars for the Chatbot, Error, and User participants in the chat.
- `cl.Avatar()` : the Avatar class allows you to display an avatar image next to a message instead of the author name. You need to send the element once. Next, if an avatar's name matches an author's name, the avatar will be automatically displayed. You must provide either a URL or a path or content bytes.
- `cl.user_session.set()` : This API call sets a value in the user session dictionary. In this case, it initializes the `message_history` in the user's session with a system content message, indicating the chat's start.

Finally, the application defines the method called whenever the user sends a new message in the chat.

```
.on_message
async def main(message: str):
    # Fetching message history from user session
    message_history = cl.user_session.get("message_history")

    # Appending user's message to message history
    message_history.append({"role": "user", "content": message})

    # Creating an empty Chainlit response message
    msg = cl.Message(content="")

    # Retry the OpenAI API call if it fails
    for attempt in range(max_retries):
        try:
            # Refresh the OpenAI security token if using Azure AD
            if openai.api_type == "azure_ad":
                refresh_openai_token()

            # Sending the message to OpenAI and streaming the response
            async for stream_resp in await openai.ChatCompletion.acreate(
                engine=engine,
                model=model,
                messages=message_history,
                temperature=temperature,
                stream=True
            ):
                if stream_resp and len(stream_resp.choices) > 0:
                    token = stream_resp.choices[0]["delta"].get("content", "")
                    await msg.stream_token(token)
            break
        # Exception handling for different types of errors during the API call (Timeout, APIError
```

Here is a detailed explanation of the function steps:

- `@cl.on_message` : The on_message decorator registers a callback function `main(message: str)` to be called when the user submits a new message in the chat. It is the main function responsible for handling the chat logic.
- `cl.user_session.get()` : This API call retrieves a value from the user's session data stored in the user_session dictionary. In this case, it fetches the `message_history` from the user's session to maintain the chat history.
- `message_history.append()` : This API call appends a new message to the `message_history` list. It is used to add the user's message and the assistant's response to the chat history.
- `cl.Message()` : This API call creates a Chainlit Message object. The `Message` class is designed to send, stream, edit, or remove messages in the chatbot user interface. In this sample, the `Message` object is used to stream the OpenAI response in the chat.
- `msg.stream_token()` : The stream_token method of the Message class streams a token to the response message. It is used to send the response from the OpenAI Chat API in chunks to ensure real-time streaming in the chat.
- `await openai.ChatCompletion.acreate()` : This API call sends a message to the OpenAI Chat API in an asynchronous mode and streams the response. It uses the provided `message_history` as context for generating the assistant's response.
- The section also includes an exception handling block that retries the OpenAI API call in case of specific errors like timeouts, API errors, connection errors, invalid requests, service unavailability, and other non-

retriable errors. You can replace this code with a general-purpose retrying library for Python like Tenacity.

Below, you can read the complete code of the application.

Skip to primary navigation
Skip to main content

```
# Import packages
import os
import sys
import time
import openai
import random
import logging
import chainlit as cl
from azure.identity import DefaultAzureCredential
from dotenv import load_dotenv
from dotenv import dotenv_values

# Load environment variables from .env file
if os.path.exists(".env"):
    load_dotenv(override = True)
    config = dotenv_values(".env")

# Read environment variables
temperature = float(os.environ.get("TEMPERATURE", 0.9))
api_base = os.getenv("AZURE_OPENAI_BASE")
api_key = os.getenv("AZURE_OPENAI_KEY")
api_type = os.environ.get("AZURE_OPENAI_TYPE", "azure")
api_version = os.environ.get("AZURE_OPENAI_VERSION", "2023-06-01-preview")
engine = os.getenv("AZURE_OPENAI_DEPLOYMENT")
model = os.getenv("AZURE_OPENAI_MODEL")
system_content = os.getenv("AZURE_OPENAI_SYSTEM_MESSAGE", "You are a helpful assistant.")
max_retries = int(os.getenv("MAX_RETRIES", 5))
backoff_in_seconds = float(os.getenv("BACKOFF_IN_SECONDS", 1))
token_refresh_interval = int(os.getenv("TOKEN_REFRESH_INTERVAL", 1800))

# Configure OpenAI
openai.api_type = api_type
openai.api_version = api_version
openai.api_base = api_base
openai.api_key = api_key

# Set default Azure credential
default_credential = DefaultAzureCredential(
) if openai.api_type ==  "azure_ad" else None

# Configure a logger
logging.basicConfig(stream = sys.stdout,
                    format = '[%(asctime)s] {%(filename)s:%(lineno)d} %(levelname)s - %(message)s
                    level = logging.INFO)
logger = logging.getLogger(__name__)

def backoff(attempt : int) -> float:
    return backoff_in_seconds * 2**attempt + random.uniform(0, 1)

# Refresh the OpenAI security token every 45 minutes
def refresh_openai_token():
    token = cl.user_session.get('openai_token')
    if token ==  None or token.expires_on < int(time.time()) - token_refresh_interval:
        cl.user_session.set('openai_token', default_credential.get_token(
            "https://cognitiveservices.azure.com/.default"))
        openai.api_key = cl.user_session.get('openai_token').token

@cl.on_chat_start
async def start_chat():
```

Skip to main content
Skip to primary navigation

```
    await cl.Avatar(
        name = "Chatbot",
        url = "https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name = "Error",
        url = "https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name = "User",
        url = "https://media.architecturaldigest.com/photos/5f241de2c850b2a36b415024/master/w_160
    ).send()
    cl.user_session.set(
        "message_history",
        [{"role": "system", "content": system_content}],
    )

@cl.on_message
async def main(message: str):
    message_history = cl.user_session.get("message_history")
    message_history.append({"role": "user", "content": message})

    # Create the Chainlit response message
    msg = cl.Message(content = "")

    # Retry the OpenAI API call if it fails
    for attempt in range(max_retries):
        try:
            # Refresh the OpenAI security token if using Azure AD
            if openai.api_type ==  "azure_ad":
                refresh_openai_token()

            # Send the message to OpenAI in an asynchronous mode and stream the response
            async for stream_resp in await openai.ChatCompletion.acreate(
                engine = engine,
                model = model,
                messages = message_history,
                temperature = temperature,
                stream = True
            ):
                if stream_resp and len(stream_resp.choices) > 0:
                    token = stream_resp.choices[0]["delta"].get("content", "")
                    await msg.stream_token(token)
            break
        except openai.error.Timeout:
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API timeout occurred. Waiting {wait_time} seconds and tryin
            time.sleep(wait_time)
        except openai.error.APIError:
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API error occurred. Waiting {wait_time} seconds and trying
            time.sleep(wait_time)
        except openai.error.APIConnectionError:
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API connection error occurred. Check your network settings,
            time.sleep(wait_time)
        except openai.error.Inval
```

Skip to primary navigation Skip to main content

```
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API invalid request. Check the documentation for the specif
            time.sleep(wait_time)
        except openai.error.ServiceUnavailableError:
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API service unavailable. Waiting {wait_time} seconds and tr
            time.sleep(wait_time)
        except Exception as e:
            logger.exception(f"A non retriable error occurred. {e}")
            break

    message_history.append({"role": "assistant", "content": msg.content})
    await msg.send()
```

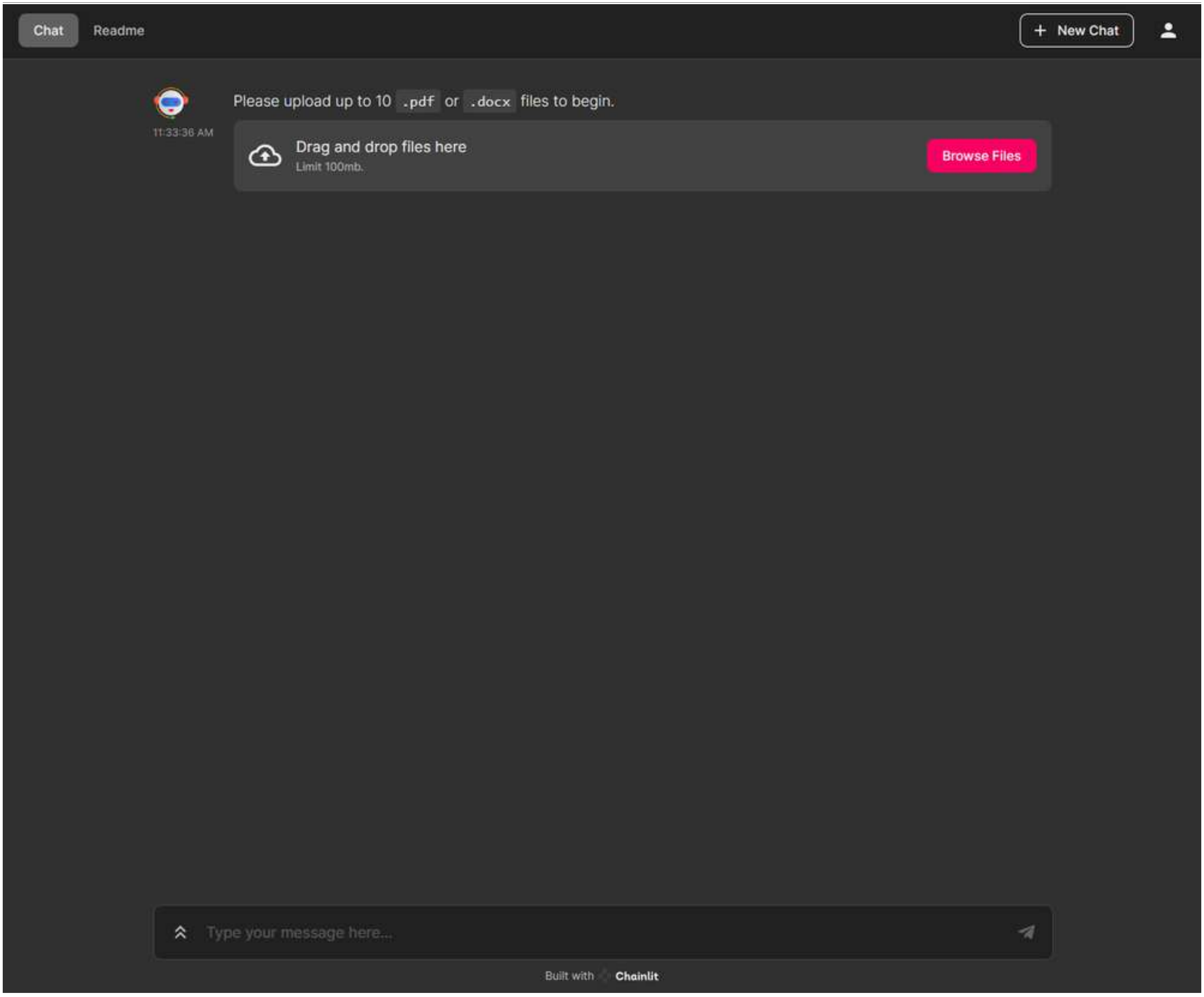
You can run the application locally using the following command. The `-w` flag` indicates auto-reload whenever we make changes live in our application code.

```
chainlit run app.py -w
```

Documents QA Chat

The Documents QA Chat application allows users to submit up to 10 `.pdf` and `.docx` documents. The application processes the uploaded documents to create vector embeddings. These embeddings are stored in ChromaDB vector database for efficient retrieval. Users can pose questions about the uploaded documents and view the Chain of Thought, enabling easy exploration of the reasoning process. The completion message contains links to the text chunks in the files that were used as a source for the response. The following picture shows the chat application interface. As you can see, you can click the `Browse` button and choose up to 10 `.pdf` and `.docx` documents to upload. Alternatively, you can drag and drop the files over the control area.

Skip to primary navigation Skip to main content

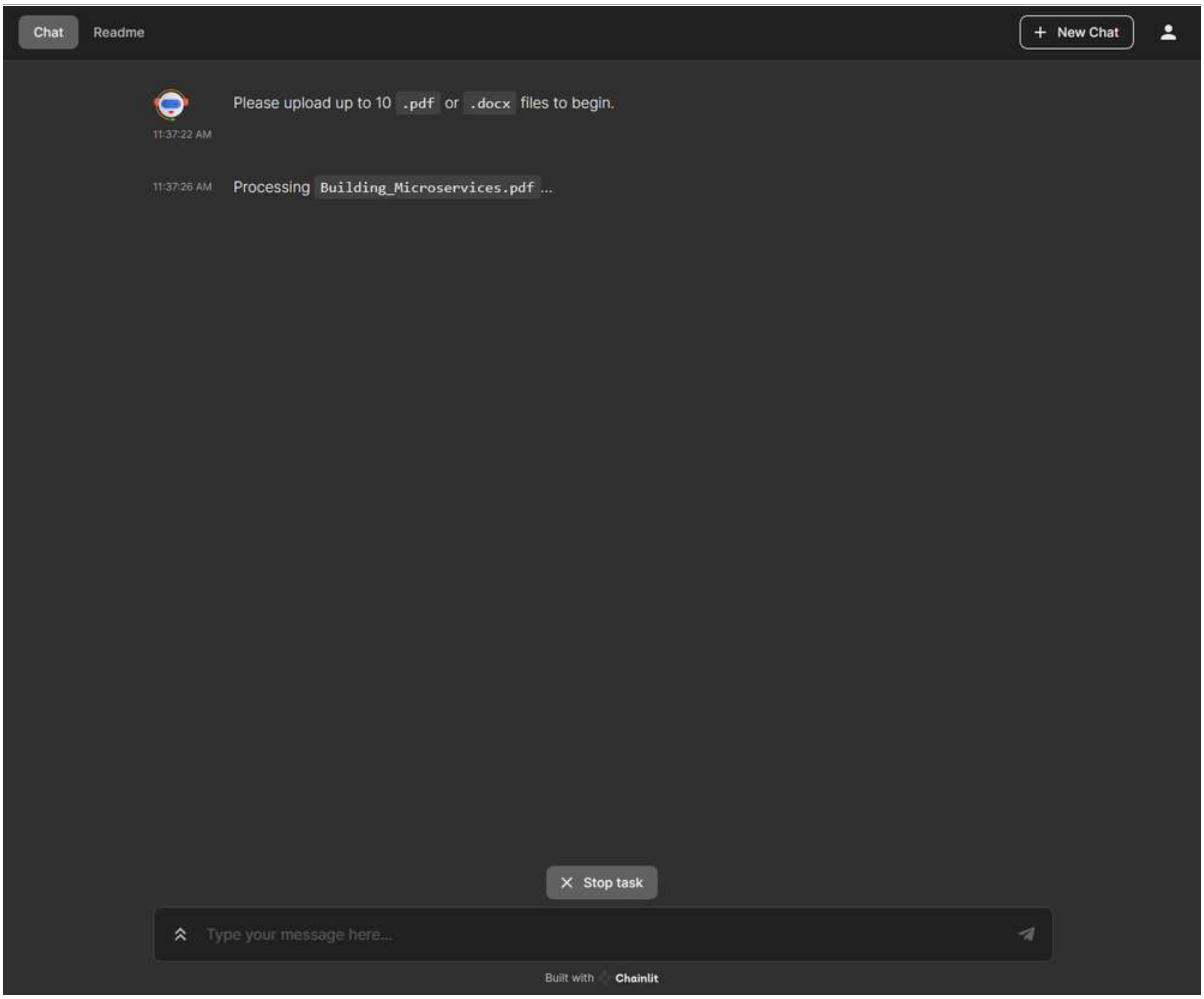


Please upload up to 10 `.pdf` or `.docx` files to begin.

Drag and drop files here
Limit 100mb.

After uploading the documents, the application creates and stores embeddings to ChromaDB vector database. During the phase, the UI shows a message `Processing <file-1>, <file-2>...`, as shown in the following picture:

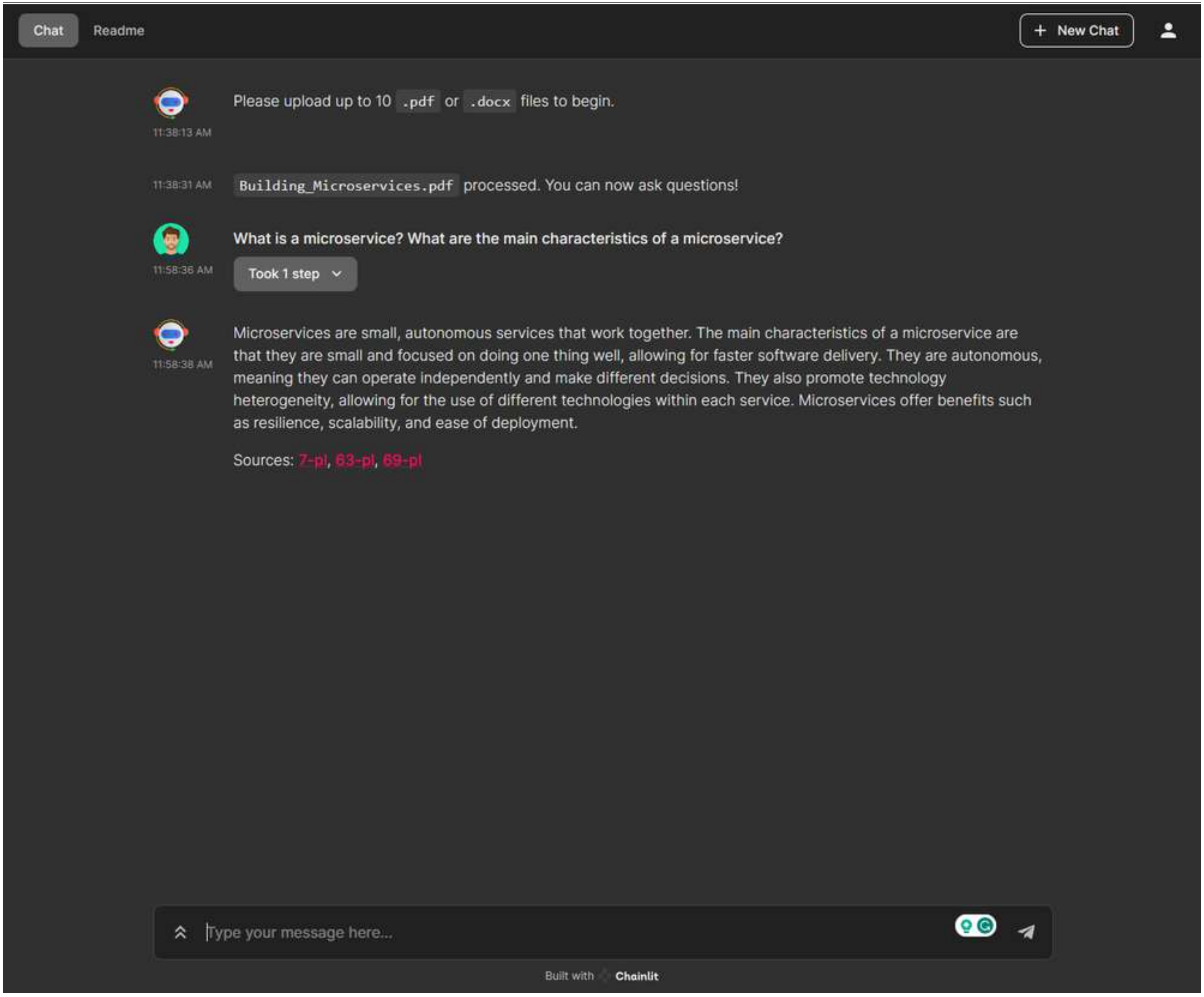
Skip to primary navigation
Skip to main content



11:37:22 AM Please upload up to 10 `.pdf` or `.docx` files to begin.

11:37:26 AM Processing `Building_Microservices.pdf` ...

When the code finished creating embeddings, the UI is ready to receive user's questions:



Please upload up to 10 `.pdf` or `.docx` files to begin.
11:38:13 AM

11:38:31 AM `Building_Microservices.pdf` processed. You can now ask questions!

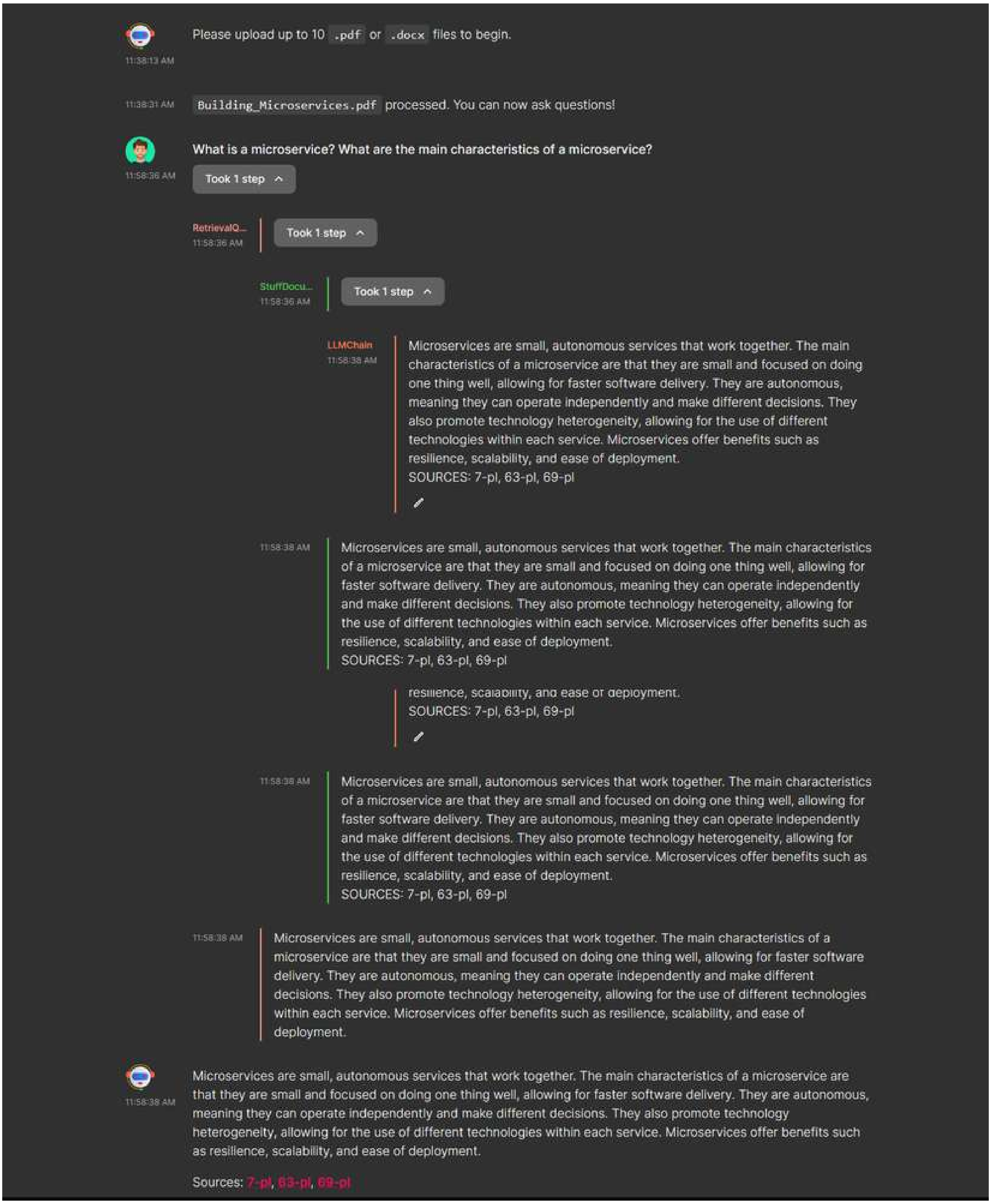
What is a microservice? What are the main characteristics of a microservice?
11:58:36 AM

Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.
11:58:38 AM

Sources: 7-pl, 63-pl, 69-pl

Understanding the individual steps for generating a specific answer can become challenging as your chat application grows in complexity. To solve this issue, Chainlit allows you to easily explore the reasoning process right from the user interface using the Chain of Thought. If you are using the LangChain integration, every intermediary step is automatically sent and displayed in the Chainlit UI just clicking and expanding the steps, as shown in the following picture:

Skip to main content
Skip to primary navigation



Please upload up to 10 `.pdf` or `.docx` files to begin.

11:38:13 AM

11:38:31 AM `Building_Microservices.pdf` processed. You can now ask questions!

What is a microservice? What are the main characteristics of a microservice?

11:58:36 AM

Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.
SOURCES: 7-pl, 63-pl, 69-pl

11:58:38 AM

Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.
SOURCES: 7-pl, 63-pl, 69-pl

resilience, scalability, and ease of deployment.
SOURCES: 7-pl, 63-pl, 69-pl

11:58:38 AM

Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.
SOURCES: 7-pl, 63-pl, 69-pl

11:58:38 AM

Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.

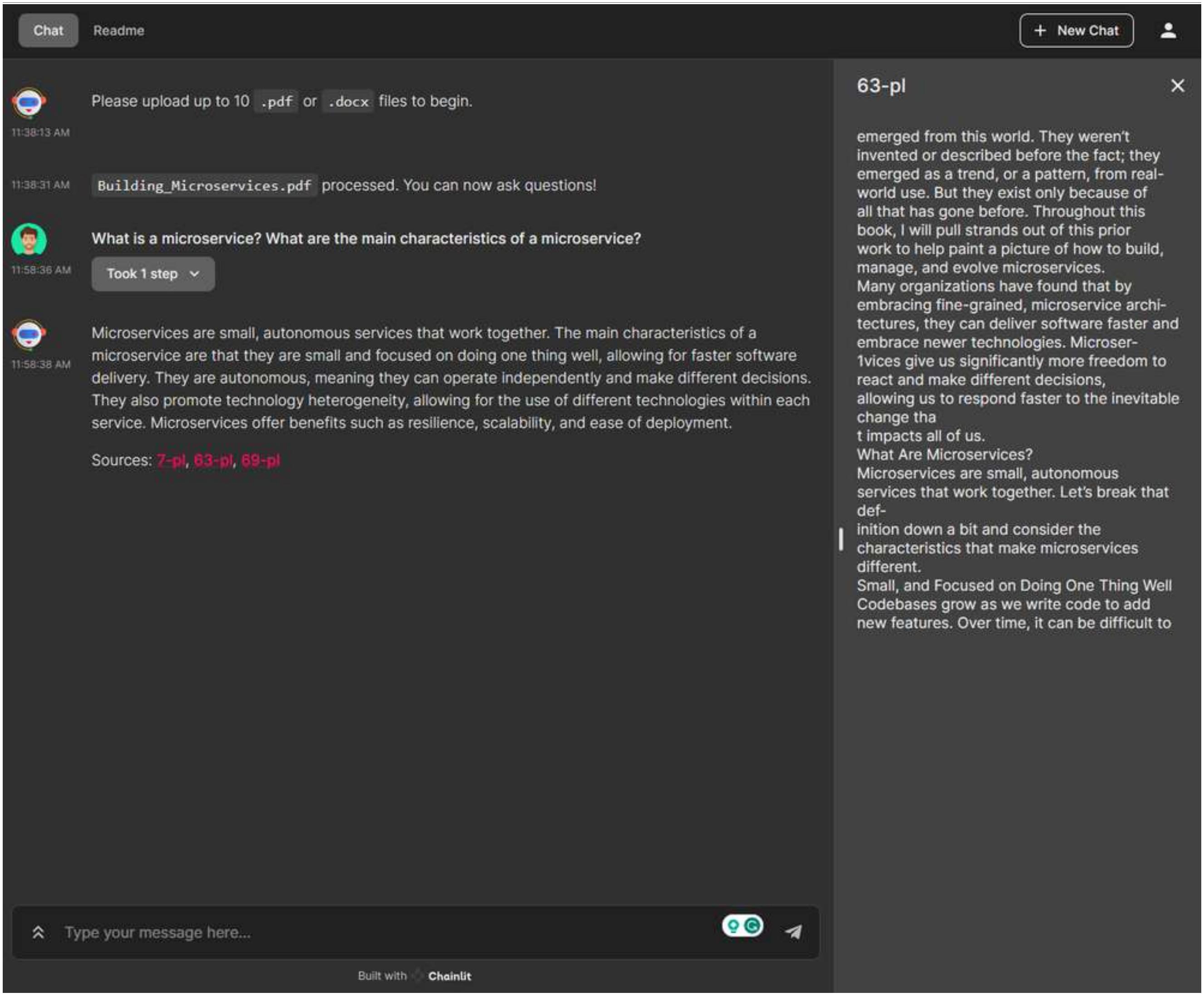
11:58:38 AM

Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.

Sources: 7-pl, 63-pl, 69-pl

To see the text chunks that the large language model used to originate the response, you can click the sources links, as shown in the following picture:

Skip to primary navigation
Skip to main content



11:38:13 AM Please upload up to 10 `.pdf` or `.docx` files to begin.

11:38:31 AM `Building_Microservices.pdf` processed. You can now ask questions!

11:58:36 AM **What is a microservice? What are the main characteristics of a microservice?**

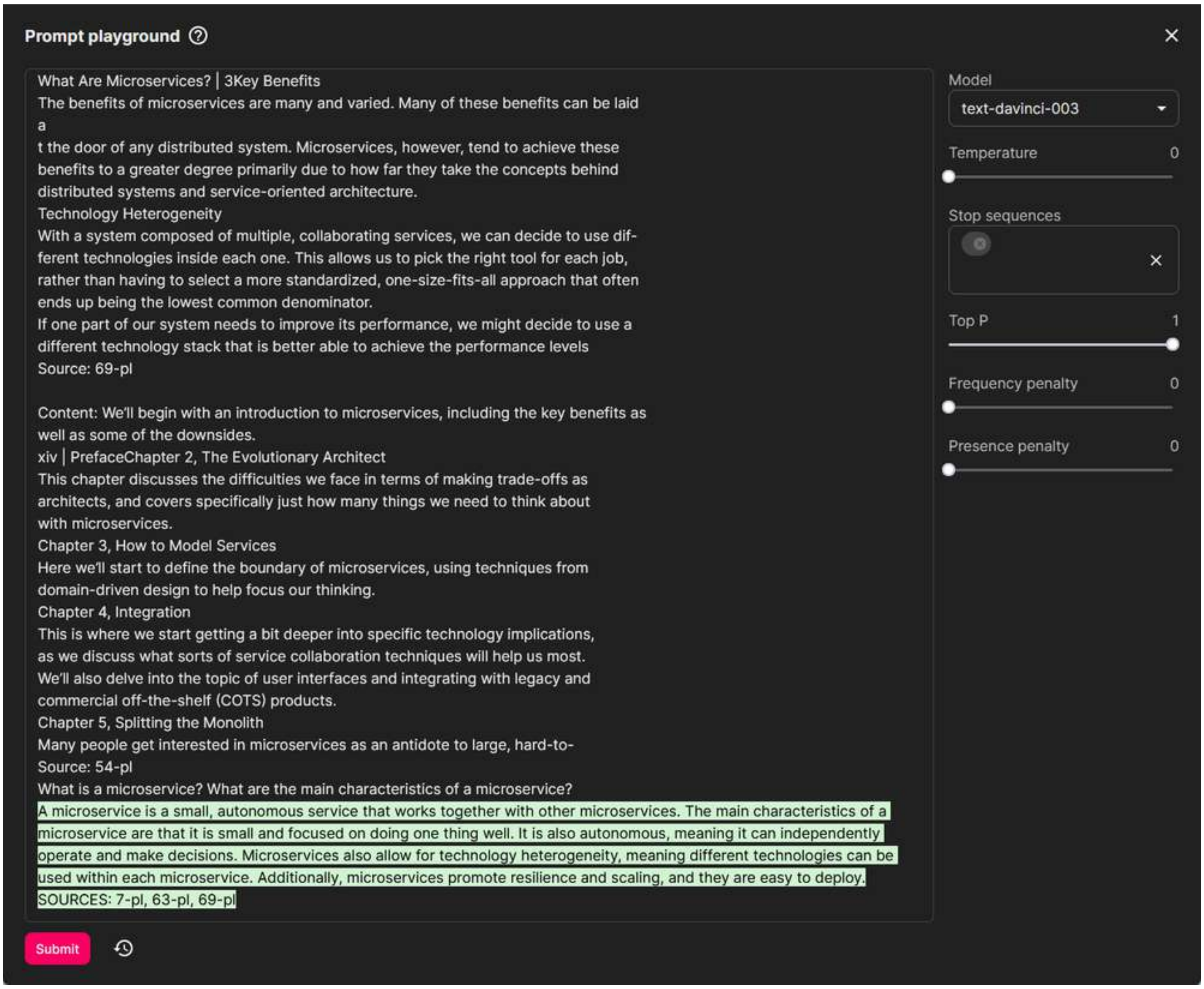
11:58:38 AM Microservices are small, autonomous services that work together. The main characteristics of a microservice are that they are small and focused on doing one thing well, allowing for faster software delivery. They are autonomous, meaning they can operate independently and make different decisions. They also promote technology heterogeneity, allowing for the use of different technologies within each service. Microservices offer benefits such as resilience, scalability, and ease of deployment.

Sources: 7-pl, 63-pl, 69-pl

emerged from this world. They weren't invented or described before the fact; they emerged as a trend, or a pattern, from real-world use. But they exist only because of all that has gone before. Throughout this book, I will pull strands out of this prior work to help paint a picture of how to build, manage, and evolve microservices.
Many organizations have found that by embracing fine-grained, microservice architectures, they can deliver software faster and embrace newer technologies. Microser-
1vices give us significantly more freedom to react and make different decisions, allowing us to respond faster to the inevitable change tha
t impacts all of us.
What Are Microservices?
Microservices are small, autonomous services that work together. Let's break that def-
inition down a bit and consider the characteristics that make microservices different.
Small, and Focused on Doing One Thing Well
Codebases grow as we write code to add new features. Over time, it can be difficult to

In the Chain of Thought, below each message, you can find an `edit` button, as a pencil icon, if a prompt generated that message. Clicking on it opens the Prompt Playground dialog, allowing you to modify and iterate on the prompt as needed.

Skip to main content
Skip to primary navigation



```
What Are Microservices? | 3Key Benefits
The benefits of microservices are many and varied. Many of these benefits can be laid
a
t the door of any distributed system. Microservices, however, tend to achieve these
benefits to a greater degree primarily due to how far they take the concepts behind
distributed systems and service-oriented architecture.
Technology Heterogeneity
With a system composed of multiple, collaborating services, we can decide to use dif-
ferent technologies inside each one. This allows us to pick the right tool for each job,
rather than having to select a more standardized, one-size-fits-all approach that often
ends up being the lowest common denominator.
If one part of our system needs to improve its performance, we might decide to use a
different technology stack that is better able to achieve the performance levels
Source: 69-pl

Content: We'll begin with an introduction to microservices, including the key benefits as
well as some of the downsides.
xiv | PrefaceChapter 2, The Evolutionary Architect
This chapter discusses the difficulties we face in terms of making trade-offs as
architects, and covers specifically just how many things we need to think about
with microservices.
Chapter 3, How to Model Services
Here we'll start to define the boundary of microservices, using techniques from
domain-driven design to help focus our thinking.
Chapter 4, Integration
This is where we start getting a bit deeper into specific technology implications,
as we discuss what sorts of service collaboration techniques will help us most.
We'll also delve into the topic of user interfaces and integrating with legacy and
commercial off-the-shelf (COTS) products.
Chapter 5, Splitting the Monolith
Many people get interested in microservices as an antidote to large, hard-to-
Source: 54-pl
What is a microservice? What are the main characteristics of a microservice?
A microservice is a small, autonomous service that works together with other microservices. The main characteristics of a
microservice are that it is small and focused on doing one thing well. It is also autonomous, meaning it can independently
operate and make decisions. Microservices also allow for technology heterogeneity, meaning different technologies can be
used within each microservice. Additionally, microservices promote resilience and scaling, and they are easy to deploy.
SOURCES: 7-pl, 63-pl, 69-pl
```

Model: text-davinci-003

Temperature: 0

Stop sequences

Top P: 1

Frequency penalty: 0

Presence penalty: 0

Let's take a look at the individual parts of the application code. The Python code starts by importing the necessary packages/modules in the following section.

Skip to primary navigation Skip to main content

```
# Import packages
import os
import io
import sys
import time
import openai
import random
import logging
import chainlit as cl
from pypdf import PdfReader
from docx import Document
from azure.identity import DefaultAzureCredential
from dotenv import load_dotenv
from dotenv import dotenv_values
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.chains import RetrievalQAWithSourcesChain
from langchain.chat_models import AzureChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

# These three lines swap the stdlib sqlite3 lib with the pysqlite3 package
__import__('pysqlite3')
sys.modules['sqlite3'] = sys.modules.pop('pysqlite3')

# Load environment variables from .env file
if os.path.exists(".env"):
    load_dotenv(override=True)
    config = dotenv_values(".env")
```

These are the libraries used by the chat application:

1. `os` : This module provides a way of interacting with the operating system, enabling the code to access environment variables, file paths, etc.
2. `sys` : This module provides access to some variables used or maintained by the interpreter and functions that interact with the interpreter.
3. `time` : This module provides various time-related functions for time manipulation and measurement.
4. `openai` : The OpenAI Python library provides convenient access to the OpenAI API from applications written in Python. It includes a pre-defined set of classes for API resources that initialize themselves dynamically from API responses, which makes it compatible with a wide range of versions of the OpenAI API. You can find usage examples for the OpenAI Python library in our API reference and the OpenAI Cookbook.
5. `random` : This module provides functions to generate random numbers.
6. `logging` : This module provides flexible logging of messages.
7. `chainlit as cl` : This imports the Chainlit library and aliases it as `cl.` Chainlit is used to create the UI of the application.
8. `DefaultAzureCredential` from `azure.identity` : when the `openai_type` property value is `azure_ad` , a `DefaultAzureCredential` object from the Azure Identity client library for Python - version 1.13.0 is

used to acquire security token from the Azure Active Directory using the credentials of the user-defined managed identity, whose client ID is defined in the `AZURE_CLIENT_ID` environment variable.

9. `load_dotenv` and `dotenv_values` from `dotenv`: Python-dotenv reads key-value pairs from a `.env` file and can set them as environment variables. It helps in the development of applications following the 12-factor principles.
10. `langchain`: Large language models (LLMs) are emerging as a transformative technology, enabling developers to build applications that they previously could not. However, using these LLMs in isolation is often insufficient for creating a truly powerful app - the real power comes when you can combine them with other sources of computation or knowledge. LangChain library aims to assist in the development of those types of applications.

The `requirements.txt` file under the `src` folder contains the list of packages used by the chat applications. You can restore these packages in your environment using the following command:

```
pip install -r requirements.txt --upgrade
```

Next, the code reads environment variables and configures the OpenAI settings.

Skip to primary navigation
Skip to main content

```
# Read environment variables
temperature = float(os.environ.get("TEMPERATURE", 0.9))
api_base = os.getenv("AZURE_OPENAI_BASE")
api_key = os.getenv("AZURE_OPENAI_KEY")
api_type = os.environ.get("AZURE_OPENAI_TYPE", "azure")
api_version = os.environ.get("AZURE_OPENAI_VERSION", "2023-06-01-preview")
chat_completion_deployment = os.getenv("AZURE_OPENAI_DEPLOYMENT")
embeddings_deployment = os.getenv("AZURE_OPENAI_ADA_DEPLOYMENT")
model = os.getenv("AZURE_OPENAI_MODEL")
max_size_mb = int(os.getenv("CHAINLIT_MAX_SIZE_MB", 100))
max_files = int(os.getenv("CHAINLIT_MAX_FILES", 10))
text_splitter_chunk_size = int(os.getenv("TEXT_SPLITTER_CHUNK_SIZE", 1000))
text_splitter_chunk_overlap = int(os.getenv("TEXT_SPLITTER_CHUNK_OVERLAP", 10))
embeddings_chunk_size = int(os.getenv("EMBEDDINGS_CHUNK_SIZE", 16))
max_retries = int(os.getenv("MAX_RETRIES", 5))
backoff_in_seconds = float(os.getenv("BACKOFF_IN_SECONDS", 1))
token_refresh_interval = int(os.getenv("TOKEN_REFRESH_INTERVAL", 1800))

# Configure system prompt
system_template = """Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
ALWAYS return a "SOURCES" part in your answer.
The "SOURCES" part should be a reference to the source of the document from which you got your an

Example of your response should be:

---

The answer is foo
SOURCES: xyz

---

Begin!
----------------
{summaries}"""
messages = [
    SystemMessagePromptTemplate.from_template(system_template),
    HumanMessagePromptTemplate.from_template("{question}"),
]
prompt = ChatPromptTemplate.from_messages(messages)
chain_type_kwargs = {"prompt": prompt}

# Configure OpenAI
openai.api_type = api_type
openai.api_version = api_version
openai.api_base = api_base
openai.api_key = api_key
```

Here's a brief explanation of each variable and related environment variable:

1. `temperature` : A float value representing the temperature for Create chat completion method of the OpenAI API. It is fetched from the environment variables with a default value of 0.9.
2. `api_base` : The base URL for the OpenAI API.
3. `api_key` : The API key for the OpenAI API.
4. `api_type` : A string representing the type of the OpenAI API.
5. `api_version` : A string representing the version of the OpenAI API.

6. `chat_completion_deployment` : the name of the Azure OpenAI GPT model for chat completion.
7. `embeddings_deployment` : the name of the Azure OpenAI deployment for embeddings.
8. `model` : The model used for chat completion calls (e.g, `gpt-35-turbo-16k`).
9. `max_size_mb` : the maximum size for the uploaded documents.
10. `max_files` : the maximum number of documents that can be uploaded.
11. `text_splitter_chunk_size` : the maximum chunk size used by the `RecursiveCharacterTextSplitter` object.
12. `text_splitter_chunk_overlap` : the maximum chunk overlap used by the `RecursiveCharacterTextSplitter` object.
13. `embeddings_chunk_size` : the maximum chunk size used by the `OpenAIEmbeddings` object.
14. `max_retries` : The maximum number of retries for OpenAI API calls.
15. `backoff_in_seconds` : The backoff time in seconds for retries in case of failures.
16. `system_template` : The content of the system message used for OpenAI API calls.

In the next section, the code sets the default Azure credential based on the `api_type` and configures a logger for logging purposes.

```
# Set default Azure credential
default_credential = DefaultAzureCredential() if openai.api_type == "azure_ad" else None

# Configure a logger
logging.basicConfig(
    stream=sys.stdout,
    format='[%(asctime)s] {%(filename)s:%(lineno)d} %(levelname)s - %(message)s',
    level=logging.INFO
)
logger = logging.getLogger(__name__)
```

Here's a brief explanation:

1. `default_credential` : It sets the default Azure credential to `DefaultAzureCredential()` if the `api_type` is "azure_ad"; otherwise, it is set to `None` .
2. `logging.basicConfig()` : This function configures the logging system with specific settings.
   - `stream` : The output stream where log messages will be written. Here, it is set to `sys.stdout` for writing log messages to the standard output.
   - `format` : The format string for log messages. It includes the timestamp, filename, line number, log level, and the actual log message.
   - `level` : The logging level. It is set to `logging.INFO` , meaning only messages with the level `INFO` and above will be logged.
3. `logger` : This creates a logger instance named after the current module (`__name__`). The logger will be used to log messages throughout the code.

Next, the code defines a helper function `backoff` that takes an integer `attempt` and returns a float value representing the backoff time for exponential retries in case of API call failures.

```
def backoff(attempt: int) -> float:
    return backoff_in_seconds * 2 ** attempt + random.uniform(0, 1)
```

The backoff time is calculated using the `backoff_in_seconds` and `attempt` variables. It follows the formula `backoff_in_seconds * 2 ** attempt + random.uniform(0, 1)` . This formula increases the backoff time exponentially with each attempt and adds a random value between 0 and 1 to avoid synchronized retries.

Then, the application defines a function called `refresh_openai_token()` to refresh the OpenAI security token if needed.

```
def refresh_openai_token():
    token = cl.user_session.get('openai_token')
    if token is None or token.expires_on < int(time.time()) - token_refresh_interval:
        cl.user_session.set('openai_token', default_credential.get_token("https://cognitiveservic
        openai.api_key = cl.user_session.get('openai_token').token
```

The function follows these steps:

1. It fetches the current token from `cl.user_session` (which seems to be a part of the `chainlit` library) using the key `'openai_token'`. The user session is a dictionary that stores the user's session data. The id and env keys are reserved for the session ID and environment variables. Other keys can be used to store arbitrary data in the user's session.
2. It checks if the token is `None` or if its expiration time (`expires_on`) is less than the current time minus 1800 seconds (30 minutes).

Next, the code defines a function called `start_chat` that is used to initialize the when the user connects to the application or clicks the `New Chat` button.

```
.on_chat_start
async def start_chat():
    # Sending Avatars for Chat Participants
    await cl.Avatar(
        name="Chatbot",
        url="https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name="Error",
        url="https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name="User",
        url="https://media.architecturaldigest.com/photos/5f241de2c850b2a36b415024/master/w_1600%
    ).send()
```

Here is a brief explanation of the function steps:

- `@cl.on_chat_start` : The on_chat_start decorator registers a callback function `start_chat()` to be called when the Chainlit chat starts. It is used to set up the chat and send avatars for the Chatbot, Error, and User participants in the chat.
- `cl.Avatar()` : the Avatar class allows you to display an avatar image next to a message instead of the author's name. You need to send the element once. Next, if an avatar's name matches an author's name, the avatar will be automatically displayed. You must provide either a URL or a path or content bytes.

The following code is used to initialize the large language model (LLM) chain used to reply to questions on the content of the uploaded documents.

```
# Initialize the file list to None
    files = None

    # Wait for the user to upload a file
    while files is None:
        files = await cl.AskFileMessage(
            content=f"Please upload up to {max_files} `.pdf` or `.docx` files to begin.",
            accept=["application/pdf", "application/vnd.openxmlformats-officedocument.wordprocess
            max_size_mb=max_size_mb,
            max_files=max_files,
            timeout=86400,
            raise_on_timeout=False
        ).send()
```

The AskFileMessage API call prompts the user to upload up to a specified number of `.pdf` or `.docx` files. The uploaded files are stored in the `files` variable. The process continues until the user uploads files. For more information, see AskFileMessage.

The following code processes each uploaded file by extracting its content.

1. The text content of each file is stored in the list `all_texts` .
2. This code performs text processing and chunking. It checks the file extension to read the file content accordingly, depending on if it's a `.pdf` or a `.docx` document.
3. The text content is split into smaller chunks using the RecursiveCharacterTextSplitter LangChain object.
4. Metadata is created for each chunk and stored in the `metadatas` list.
5. If `openai.api_type == "azure_ad"` , the code invokes the `refresh_openai_token()` that gets a security token from Azure AD to communicate with the Azure OpenAI Service.

```
# Create a message to inform the user that the files are being processed
    content = ''
    if (len(files)  ==  1):
        content = f"Processing `{files[0].name}`..."
    else:
        files_names = [f"`{f.name}`" for f in files]
        content = f"Processing {', '.join(files_names)}..."
    msg = cl.Message(content = content, author = "Chatbot")
    await msg.send()

    # Create a list to store the texts of each file
    all_texts = []

    # Process each file uploaded by the user
    for file in files:

        # Create an in-memory buffer from the file content
        bytes = io.BytesIO(file.content)

        # Get file extension
        extension = file.name.split('.')[-1]

        # Initialize the text variable
        text = ''

        # Read the file
        if extension == "pdf":
            # ...
        elif extension == "docx":
            # ...

        # Split the text into chunks
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=text_splitter_chunk_size,
            chunk_overlap=text_splitter_chunk_overlap
        )
        texts = text_splitter.split_text(text)

        # Add the chunks and metadata to the list
        all_texts.extend(texts)

    # Create a metadata for each chunk
    metadatas = [{"source": f"{i}-pl"} for i in range(len(all_texts))]

    #  Refresh the OpenAI security token if using Azure AD
    if openai.api_type == "azure_ad":
        refresh_openai_token()
```

The next piece of code performs the following steps:

1. It creates an OpenAIEmbeddings configured to use the embeddings model in the Azure OpenAI Service to create embeddings from text chunks.
2. It creates a ChromaDB vector database using the `OpenAIEmbeddings` object, the text chunks list, and the metadata list.
3. It creates an AzureChatOpenAI LangChain object based on the GPR model hosted in Azure OpenAI Service.

Skip to main content Skip to primary navigation

4. It creates a chain using the RetrievalQAWithSourcesChain.from_chain_type API call uses previously created models and stores them as retrievers.
5. It stores the metadata and text chunks in the user session using the `cl.user_session.set()` API call.
6. It creates a message to inform the user that the files are ready for queries, and finally returns the `chain` .
7. The `cl.user_session.set("chain", chain)` call stores the LLM chain in the user_session dictionary for later use.

Skip to primary navigation
Skip to main content

```
#  Refresh the OpenAI security token if using Azure AD
    if openai.api_type  ==  "azure_ad":
        refresh_openai_token()

    # Create a Chroma vector store
    embeddings = OpenAIEmbeddings(
        deployment = embeddings_deployment,
        openai_api_key = openai.api_key,
        openai_api_base = openai.api_base,
        openai_api_version = openai.api_version,
        openai_api_type = openai.api_type,
        chunk_size = embeddings_chunk_size)

    # Create a Chroma vector store
    db = await cl.make_async(Chroma.from_texts)(
        all_texts, embeddings, metadatas = metadatas
    )

    # Create an AzureChatOpenAI llm
    llm = AzureChatOpenAI(
        temperature = temperature,
        openai_api_key = openai.api_key,
        openai_api_base = openai.api_base,
        openai_api_version = openai.api_version,
        openai_api_type = openai.api_type,
        deployment_name = chat_completion_deployment)

    # Create a chain that uses the Chroma vector store
    chain = RetrievalQAWithSourcesChain.from_chain_type(
        llm = llm,
        chain_type = "stuff",
        retriever = db.as_retriever(),
        return_source_documents = True,
        chain_type_kwargs = chain_type_kwargs
    )

    # Save the metadata and texts in the user session
    cl.user_session.set("metadatas", metadatas)
    cl.user_session.set("texts", all_texts)

    # Create a message to inform the user that the files are ready for queries
    content = ''
    if (len(files)  ==  1):
        content = f"`{files[0].name}` processed. You can now ask questions!"
    else:
        files_names = [f"`{f.name}`" for f in files]
        content = f"{', '.join(files_names)} processed. You can now ask questions."
    msg.content = content
    msg.author = "Chatbot"
    await msg.update()

    # Store the chain in the user session
    cl.user_session.set("chain", chain)
```

The following code handles the communication with the OpenAI API and incorporates retrying logic in case the API calls fail due to specific errors.

- `@cl.on_message` : The on_message decorator registers a callback function `main(message: str)` to be called when the user submits a new message in the chatbot. It is the main function responsible for handling the chat logic.

- `cl.user_session.get("chain")` : this call retrieves the LLM chain from the user_session dictionary.
- The `for` loop allows multiple attempts, up to `max_retries` , to communicate with the chat completion API and handles different types of API errors, such as timeout, connection error, invalid request, and service unavailability.
- `await chain.acall` : The asynchronous call to the RetrievalQAWithSourcesChain.acall executes the LLM chain with the user message as an input.

```
.on_message
async def run(message: str):
    # Retrieve the chain from the user session
    chain = cl.user_session.get("chain")

    # Initialize the response
    response =  None

    # Retry the OpenAI API call if it fails
    for attempt in range(max_retries):
        try:
            # Refresh the OpenAI security token if using Azure AD
            if openai.api_type == "azure_ad":
                refresh_openai_token()

            # Ask the question to the chain
            response = await chain.acall(message, callbacks=[cl.AsyncLangchainCallbackHandler()])
            break
        except openai.error.Timeout:
            # Exception handling for timeout error
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API timeout occurred. Waiting {wait_time} seconds and tryin
            time.sleep(wait_time)
        except openai.error.APIError:
            # Exception handling for API error
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API error occurred. Waiting {wait_time} seconds and trying
            time.sleep(wait_time)
        except openai.error.APIConnectionError:
            # Exception handling for API connection error
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API connection error occurred. Check your network settings,
            time.sleep(wait_time)
        except openai.error.InvalidRequestError:
            # Exception handling for invalid request error
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API invalid request. Check the documentation for the specif
            time.sleep(wait_time)
        except openai.error.ServiceUnavailableError:
            # Exception handling for service unavailable error
            # Implement exponential backoff
            wait_time = backoff(attempt)
            logger.exception(f"OpenAI API service unavailable. Waiting {wait_time} seconds and tr
            time.sleep(wait_time)
        except Exception as e:
            # Exception handling for non-retriable errors
            logger.exception(f"A non-retriable error occurred. {e}")
            break
```

The code below extracts the answers and sources from the API response and formats them to be sent as a message.

- The `answer` and `sources` are obtained from the `response` dictionary.

Skip to main content

- The sources are then processed to find corresponding texts in the user session metadata (`metadatas`) and create `source_elements` using `cl.Text()` .
- `cl.Message().send()` : the Message API creates and displays a message containing the answer and sources, if available.

```
# Get the answer and sources from the response
    answer = response["answer"]
    sources = response["sources"].strip()
    source_elements = []

    # Get the metadata and texts from the user session
    metadatas = cl.user_session.get("metadatas")
    all_sources = [m["source"] for m in metadatas]
    texts = cl.user_session.get("texts")

    if sources:
        found_sources = []

        # Add the sources to the message
        for source in sources.split(","):
            source_name = source.strip().replace(".", "")
            # Get the index of the source
            try:
                index = all_sources.index(source_name)
            except ValueError:
                continue
            text = texts[index]
            found_sources.append(source_name)
            # Create the text element referenced in the message
            source_elements.append(cl.Text(content=text, name=source_name))

        if found_sources:
            answer += f"\nSources: {', '.join(found_sources)}"
        else:
            answer += "\nNo sources found"

    await cl.Message(content=answer, elements=source_elements).send()
```

Below, you can read the complete code of the application.

```
# Import packages
import os
import io
import sys
import time
import openai
import random
import logging
import chainlit as cl
from pypdf import PdfReader
from docx import Document
from azure.identity import DefaultAzureCredential
from dotenv import load_dotenv
from dotenv import dotenv_values
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.chains import RetrievalQAWithSourcesChain
from langchain.chat_models import AzureChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

# These three lines swap the stdlib sqlite3 lib with the pysqlite3 package
__import__('pysqlite3')
sys.modules['sqlite3'] = sys.modules.pop('pysqlite3')

# Load environment variables from .env file
if os.path.exists(".env"):
    load_dotenv(override = True)
    config = dotenv_values(".env")

# Read environment variables
temperature = float(os.environ.get("TEMPERATURE", 0.9))
api_base = os.getenv("AZURE_OPENAI_BASE")
api_key = os.getenv("AZURE_OPENAI_KEY")
api_type = os.environ.get("AZURE_OPENAI_TYPE", "azure")
api_version = os.environ.get("AZURE_OPENAI_VERSION", "2023-06-01-preview")
chat_completion_deployment = os.getenv("AZURE_OPENAI_DEPLOYMENT")
embeddings_deployment = os.getenv("AZURE_OPENAI_ADA_DEPLOYMENT")
model = os.getenv("AZURE_OPENAI_MODEL")
max_size_mb = int(os.getenv("CHAINLIT_MAX_SIZE_MB", 100))
max_files = int(os.getenv("CHAINLIT_MAX_FILES", 10))
text_splitter_chunk_size = int(os.getenv("TEXT_SPLITTER_CHUNK_SIZE", 1000))
text_splitter_chunk_overlap = int(os.getenv("TEXT_SPLITTER_CHUNK_OVERLAP", 10))
embeddings_chunk_size = int(os.getenv("EMBEDDINGS_CHUNK_SIZE", 16))
max_retries = int(os.getenv("MAX_RETRIES", 5))
backoff_in_seconds = float(os.getenv("BACKOFF_IN_SECONDS", 1))

# Configure system prompt
system_template = """Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
ALWAYS return a "SOURCES" part in your answer.
The "SOURCES" part should be a reference to the source of the document from which you got your an

Example of your response should be:
```

Skip to primary navigation
Skip to main content

```
---

The answer is foo
SOURCES: xyz

---

Begin!
----------------
{summaries}"""
messages = [
    SystemMessagePromptTemplate.from_template(system_template),
    HumanMessagePromptTemplate.from_template("{question}"),
]
prompt = ChatPromptTemplate.from_messages(messages)
chain_type_kwargs = {"prompt": prompt}

# Configure OpenAI
openai.api_type = api_type
openai.api_version = api_version
openai.api_base = api_base
openai.api_key = api_key

# Set default Azure credential
default_credential = DefaultAzureCredential(
) if openai.api_type  ==  "azure_ad" else None

# Configure a logger
logging.basicConfig(stream = sys.stdout,
                    format = '[%(asctime)s] {%(filename)s:%(lineno)d} %(levelname)s - %(message)s
                    level = logging.INFO)
logger = logging.getLogger(__name__)

# Refresh the OpenAI security token every 45 minutes
def refresh_openai_token():
    token = cl.user_session.get('openai_token')
    if token  ==  None or token.expires_on < int(time.time()) - 1800:
        cl.user_session.set('openai_token', default_credential.get_token(
            "https://cognitiveservices.azure.com/.default"))
        openai.api_key = cl.user_session.get('openai_token').token

def backoff(attempt : int) -> float:
    return backoff_in_seconds * 2**attempt + random.uniform(0, 1)

@cl.on_chat_start
async def start():
    await cl.Avatar(
        name = "Chatbot",
        url = "https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name = "Error",
        url = "https://cdn-icons-png.flaticon.com/512/8649/8649595.png"
    ).send()
    await cl.Avatar(
        name = "User",
        url = "https://media.architecturaldigest.com/photos/5f241de2c850b2a36b415024/master/w_160
    ).send()

    # Initialize the file list to
```

Skip to primary navigation
Skip to main content

```
files = None

# Wait for the user to upload a file
while files  ==  None:
    files = await cl.AskFileMessage(
        content = f"Please upload up to {max_files} `.pdf` or `.docx` files to begin.",
        accept = ["application/pdf",
                "application/vnd.openxmlformats-officedocument.wordprocessingml.document"],
        max_size_mb = max_size_mb,
        max_files = max_files,
        timeout = 86400,
        raise_on_timeout = False
    ).send()

# Create a message to inform the user that the files are being processed
content = ''
if (len(files)  ==  1):
    content = f"Processing `{files[0].name}`..."
else:
    files_names = [f"`{f.name}`" for f in files]
    content = f"Processing {', '.join(files_names)}..."
msg = cl.Message(content = content, author = "Chatbot")
await msg.send()

# Create a list to store the texts of each file
all_texts = []

# Process each file uplodaded by the user
for file in files:

    # Create an in-memory buffer from the file content
    bytes = io.BytesIO(file.content)

    # Get file extension
    extension = file.name.split('.')[-1]

    # Initialize the text variable
    text = ''

    # Read the file
    if extension  ==  "pdf":
        reader = PdfReader(bytes)
        for i in range(len(reader.pages)):
            text +=  reader.pages[i].extract_text()
    elif extension  ==  "docx":
        doc = Document(bytes)
        paragraph_list = []
        for paragraph in doc.paragraphs:
            paragraph_list.append(paragraph.text)
        text = '\n'.join(paragraph_list)

    # Split the text into chunks
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size = text_splitter_chunk_size,
        chunk_overlap = text_splitter_chunk_overlap)
    texts = text_splitter.split_text(text)

    # Add the chunks and metadata to the list
    all_texts.extend(texts)
```

Skip to primary navigation
Skip to main content

```
    # Create a metadata for each chunk
    metadatas = [{"source": f"{i}-pl"} for i in range(len(all_texts))]

    #  Refresh the OpenAI security token if using Azure AD
    if openai.api_type  ==  "azure_ad":
        refresh_openai_token()

    # Create a Chroma vector store
    embeddings = OpenAIEmbeddings(
        deployment = embeddings_deployment,
        openai_api_key = openai.api_key,
        openai_api_base = openai.api_base,
        openai_api_version = openai.api_version,
        openai_api_type = openai.api_type,
        chunk_size = embeddings_chunk_size)

    # Create a Chroma vector store
    db = await cl.make_async(Chroma.from_texts)(
        all_texts, embeddings, metadatas = metadatas
    )

    # Create an AzureChatOpenAI llm
    llm = AzureChatOpenAI(
        temperature = temperature,
        openai_api_key = openai.api_key,
        openai_api_base = openai.api_base,
        openai_api_version = openai.api_version,
        openai_api_type = openai.api_type,
        deployment_name = chat_completion_deployment)

    # Create a chain that uses the Chroma vector store
    chain = RetrievalQAWithSourcesChain.from_chain_type(
        llm = llm,
        chain_type = "stuff",
        retriever = db.as_retriever(),
        return_source_documents = True,
        chain_type_kwargs = chain_type_kwargs
    )

    # Save the metadata and texts in the user session
    cl.user_session.set("metadatas", metadatas)
    cl.user_session.set("texts", all_texts)

    # Create a message to inform the user that the files are ready for queries
    content = ''
    if (len(files)  ==  1):
        content = f"`{files[0].name}` processed. You can now ask questions!"
    else:
        files_names = [f"`{f.name}`" for f in files]
        content = f"{', '.join(files_names)} processed. You can now ask questions."
    msg.content = content
    msg.author = "Chatbot"
    await msg.update()

     # Store the chain in the user session
    cl.user_session.set("chain", chain)

@cl.on_message
async def run(message: str):
    # Retrieve the chain from the
```

Skip to main content
Skip to primary navigation

```
chain = cl.user_session.get("chain")

# Initialize the response
response =  None

# Retry the OpenAI API call if it fails
for attempt in range(max_retries):
    try:
        # Refresh the OpenAI security token if using Azure AD
        if openai.api_type  ==  "azure_ad":
            refresh_openai_token()

        # Ask the question to the chain
        response = await chain.acall(message, callbacks = [cl.AsyncLangchainCallbackHandler()
        break
    except openai.error.Timeout:
        # Implement exponential backoff
        wait_time = backoff(attempt)
        logger.exception(f"OpenAI API timeout occurred. Waiting {wait_time} seconds and tryin
        time.sleep(wait_time)
    except openai.error.APIError:
        # Implement exponential backoff
        wait_time = backoff(attempt)
        logger.exception(f"OpenAI API error occurred. Waiting {wait_time} seconds and trying
        time.sleep(wait_time)
    except openai.error.APIConnectionError:
        # Implement exponential backoff
        wait_time = backoff(attempt)
        logger.exception(f"OpenAI API connection error occurred. Check your network settings,
        time.sleep(wait_time)
    except openai.error.InvalidRequestError:
        # Implement exponential backoff
        wait_time = backoff(attempt)
        logger.exception(f"OpenAI API invalid request. Check the documentation for the specif
        time.sleep(wait_time)
    except openai.error.ServiceUnavailableError:
        # Implement exponential backoff
        wait_time = backoff(attempt)
        logger.exception(f"OpenAI API service unavailable. Waiting {wait_time} seconds and tr
        time.sleep(wait_time)
    except Exception as e:
        logger.exception(f"A non retriable error occurred. {e}")
        break

# Get the answer and sources from the response
answer = response["answer"]
sources = response["sources"].strip()
source_elements = []

# Get the metadata and texts from the user session
metadatas = cl.user_session.get("metadatas")
all_sources = [m["source"] for m in metadatas]
texts = cl.user_session.get("texts")

if sources:
    found_sources = []

    # Add the sources to the message
    for source in sources.split(","):
        source_name = source.strip()
```

```
            # Get the index of the source
            try:
                index = all_sources.index(source_name)
            except ValueError:
                continue
            text = texts[index]
            found_sources.append(source_name)
            # Create the text element referenced in the message
            source_elements.append(cl.Text(content = text, name = source_name))

        if found_sources:
            answer +=  f"\nSources: {', '.join(found_sources)}"
        else:
            answer +=  "\nNo sources found"

    await cl.Message(content = answer, elements = source_elements).send()
```

You can run the application locally using the following command. The `-w` flag` indicates auto-reload whenever we make changes live in our application code.

```
chainlit run app.py -w
```

Build Docker Images

You can use the `src/01-build-docker-images.sh` Bash script to build the Docker container image for each container app.

```
#!/bin/bash

# Variables
source ./00-variables.sh

# Use a for loop to build the docker images using the array index
for index in ${!images[@]}; do
  # Build the docker image
  docker build -t ${images[$index]}:$tag -f Dockerfile --build-arg FILENAME=${filenames[$index]}
done
```

Before running any script in the `src` folder, make sure to customize the value of the variables inside the `00-variables.sh` file located in the same folder. This file is embedded in all the scripts and contains the following variables:

```
# Variables

# Azure Container Registry
prefix="Blue"
acrName="${prefix}Registry"
acrResourceGrougName="${prefix}RG"
location="EastUS"

# Python Files
docAppFile="doc.py"
chatAppFile="chat.py"

# Docker Images
docImageName="doc"
chatImageName="chat"
tag="v1"
port="8000"

# Arrays
images=($docImageName $chatImageName)
filenames=($docAppFile $chatAppFile)
```

The `Dockerfile` under the `src` folder is parametric and can be used to build the container images for both chat applications.

Skip to main content
Skip to primary navigation

```
# app/Dockerfile

# # Stage 1 - Install build dependencies

# A Dockerfile must start with a FROM instruction that sets the base image for the container.
# The Python images come in many flavors, each designed for a specific use case.
# The python:3.11-slim image is a good base image for most applications.
# It is a minimal image built on top of Debian Linux and includes only the necessary packages to
# The slim image is a good choice because it is small and contains only the packages needed to ru
# For more information, see:
# * https://hub.docker.com/_/python
# * https://docs.streamlit.io/knowledge-base/tutorials/deploy/docker
FROM python:3.11-slim AS builder

# The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD i
# If the WORKDIR doesn’t exist, it will be created even if it’s not used in any subsequent Docker
# For more information, see: https://docs.docker.com/engine/reference/builder/#workdir
WORKDIR /app

# Set environment variables.
# The ENV instruction sets the environment variable <key> to the value <value>.
# This value will be in the environment of all “descendant” Dockerfile commands and can be replac
# For more information, see: https://docs.docker.com/engine/reference/builder/#env
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Install git so that we can clone the app code from a remote repo using the RUN instruction.
# The RUN comand has 2 forms:
# * RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on L
# * RUN ["executable", "param1", "param2"] (exec form)
# The RUN instruction will execute any commands in a new layer on top of the current image and co
# The resulting committed image will be used for the next step in the Dockerfile.
# For more information, see: https://docs.docker.com/engine/reference/builder/#run
RUN apt-get update && apt-get install -y \
  build-essential \
  curl \
  software-properties-common \
  git \
  && rm -rf /var/lib/apt/lists/*

# Create a virtualenv to keep dependencies together
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

# Clone the requirements.txt which contains dependencies to WORKDIR
# COPY has two forms:
# * COPY <src> <dest> (this copies the files from the local machine to the container's own filesy
# * COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)
# For more information, see: https://docs.docker.com/engine/reference/builder/#copy
COPY requirements.txt .

# Install the Python dependencies
RUN pip install --no-cache-dir --no-deps -r requirements.txt

# Stage 2 - Copy only necessary files to the runner stage

# The FROM instruction initializes a new build stage for the application
FROM python:3.11-slim
```

Skip to primary navigation
Skip to main content

```
# Define the filename to copy as an argument
ARG FILENAME

# Deefine the port to run the application on as an argument
ARG PORT=8000

# Set an environment variable
ENV FILENAME=${FILENAME}

# Sets the working directory to /app
WORKDIR /app

# Copy the virtual environment from the builder stage
COPY --from=builder /opt/venv /opt/venv

# Set environment variables
ENV PATH="/opt/venv/bin:$PATH"

# Clone the $FILENAME containing the application code
COPY $FILENAME .

# Copy the chainlit.md file to the working directory
COPY chainlit.md .

# Copy the .chainlit folder to the working directory
COPY ./.chainlit ./.chainlit

# The EXPOSE instruction informs Docker that the container listens on the specified network ports
# For more information, see: https://docs.docker.com/engine/reference/builder/#expose
EXPOSE $PORT

# The ENTRYPOINT instruction has two forms:
# * ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)
# * ENTRYPOINT command param1 param2 (shell form)
# The ENTRYPOINT instruction allows you to configure a container that will run as an executable.
# For more information, see: https://docs.docker.com/engine/reference/builder/#entrypoint
CMD chainlit run $FILENAME --port=$PORT
```

Test applications locally

You can use the `src/02-run-docker-container.sh` Bash script to test the containers for the `sender`, `processor`, and `receiver` applications.

```bash
#!/bin/bash

# Variables
source ./00-variables.sh

# Print the menu
echo "===================================="
echo "Run Docker Container (1-3): "
echo "===================================="
options=(
  "Doc"
  "Chat"
)
name=""
# Select an option
COLUMNS=0
select option in "${options[@]}"; do
  case $option in
    "Doc")
      docker run -it \
      --rm \
      -p $port:$port \
      -e AZURE_OPENAI_BASE=$AZURE_OPENAI_BASE \
      -e AZURE_OPENAI_KEY=$AZURE_OPENAI_KEY \
      -e AZURE_OPENAI_MODEL=$AZURE_OPENAI_MODEL \
      -e AZURE_OPENAI_DEPLOYMENT=$AZURE_OPENAI_DEPLOYMENT \
      -e AZURE_OPENAI_ADA_DEPLOYMENT=$AZURE_OPENAI_ADA_DEPLOYMENT \
      -e AZURE_OPENAI_VERSION=$AZURE_OPENAI_VERSION \
      -e AZURE_OPENAI_TYPE=$AZURE_OPENAI_TYPE \
      -e TEMPERATURE=$TEMPERATURE \
      --name $docImageName \
      $docImageName:$tag
      break
    ;;
    "Chat")
      docker run -it \
      --rm \
      -p $port:$port \
      -e AZURE_OPENAI_BASE=$AZURE_OPENAI_BASE \
      -e AZURE_OPENAI_KEY=$AZURE_OPENAI_KEY \
      -e AZURE_OPENAI_MODEL=$AZURE_OPENAI_MODEL \
      -e AZURE_OPENAI_DEPLOYMENT=$AZURE_OPENAI_DEPLOYMENT \
      -e AZURE_OPENAI_VERSION=$AZURE_OPENAI_VERSION \
      -e AZURE_OPENAI_TYPE=$AZURE_OPENAI_TYPE \
      -e TEMPERATURE=$TEMPERATURE \
      --name $chatImageName \
      $chatImageName:$tag
      break
    ;;
    "Quit")
      exit
    ;;
    *) echo "invalid option $REPLY" ;;
  esac
done
```

Skip to primary navigation
Skip to main content

Push Docker containers to the Azure Container Registry

You can use the `src/03-push-docker-image.sh` Bash script to push the Docker container images for the `sender`, `processor`, and `receiver` applications to the Azure Container Registry (ACR)

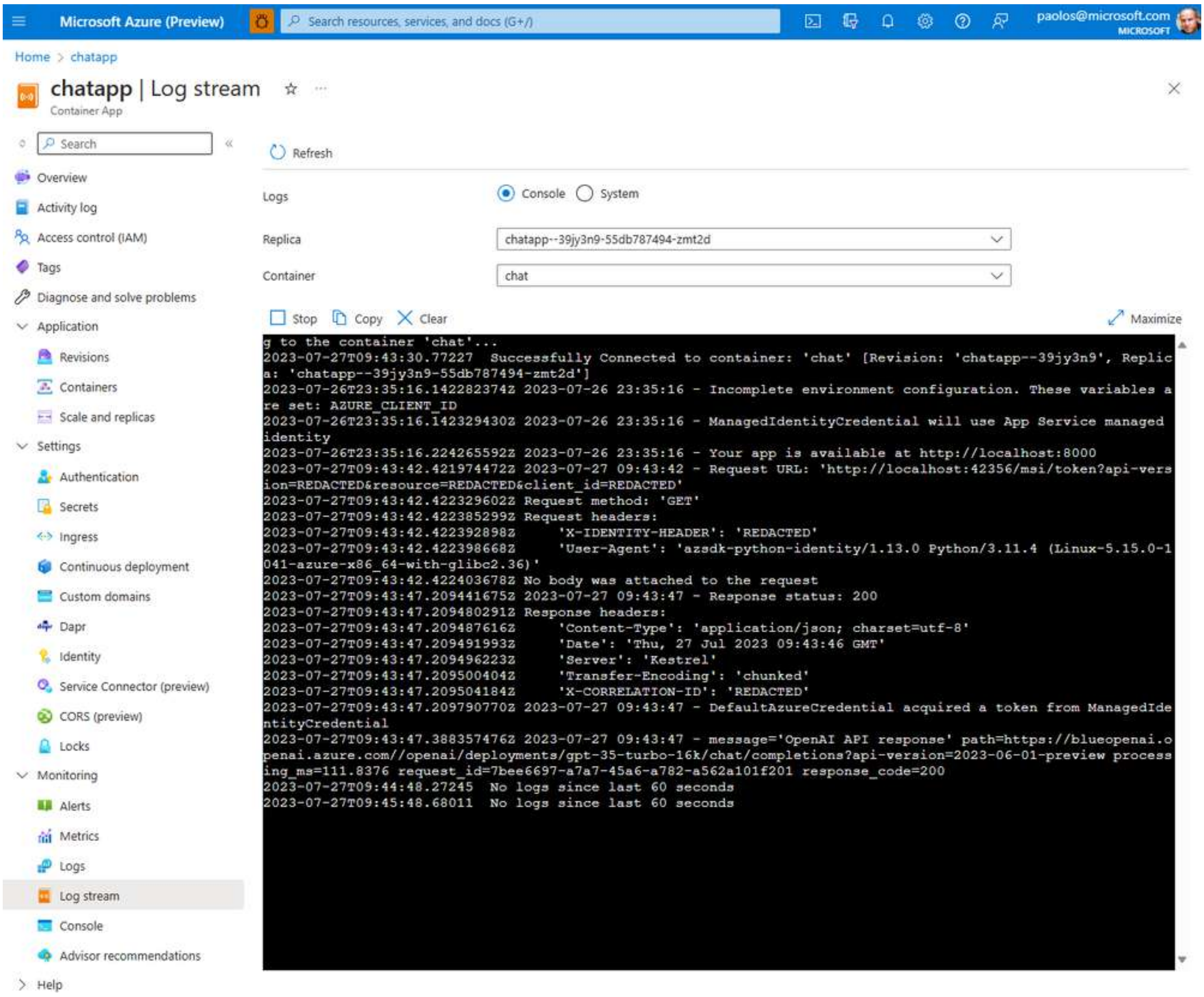
```
#!/bin/bash

# Variables
source ./00-variables.sh

# Login to ACR
echo "Logging in to [${acrName,,}] container registry..."
az acr login --name ${acrName,,}

# Retrieve ACR login server. Each container image needs to be tagged with the loginServer name of
echo "Retrieving login server for the [${acrName,,}] container registry..."
loginServer=$(az acr show --name ${acrName,,} --query loginServer --output tsv)

# Use a for loop to tag and push the local docker images to the Azure Container Registry
for index in ${!images[@]}; do
  # Tag the local sender image with the loginServer of ACR
  docker tag ${images[$index],,}:$tag $loginServer/${images[$index],,}:$tag

  # Push the container image to ACR
  docker push $loginServer/${images[$index],,}:$tag
done
```

Monitoring

Azure Container Apps provides several built-in observability features that together give you a holistic view of your container app’s health throughout its application lifecycle. These features help you monitor and diagnose the state of your app to improve performance and respond to trends and critical problems.

You can use the `Log Stream` panel on the Azure Portal to see the logs generated by a container app, as shown in the following screenshot.

Skip to primary navigation
Skip to main content
Skip to footer



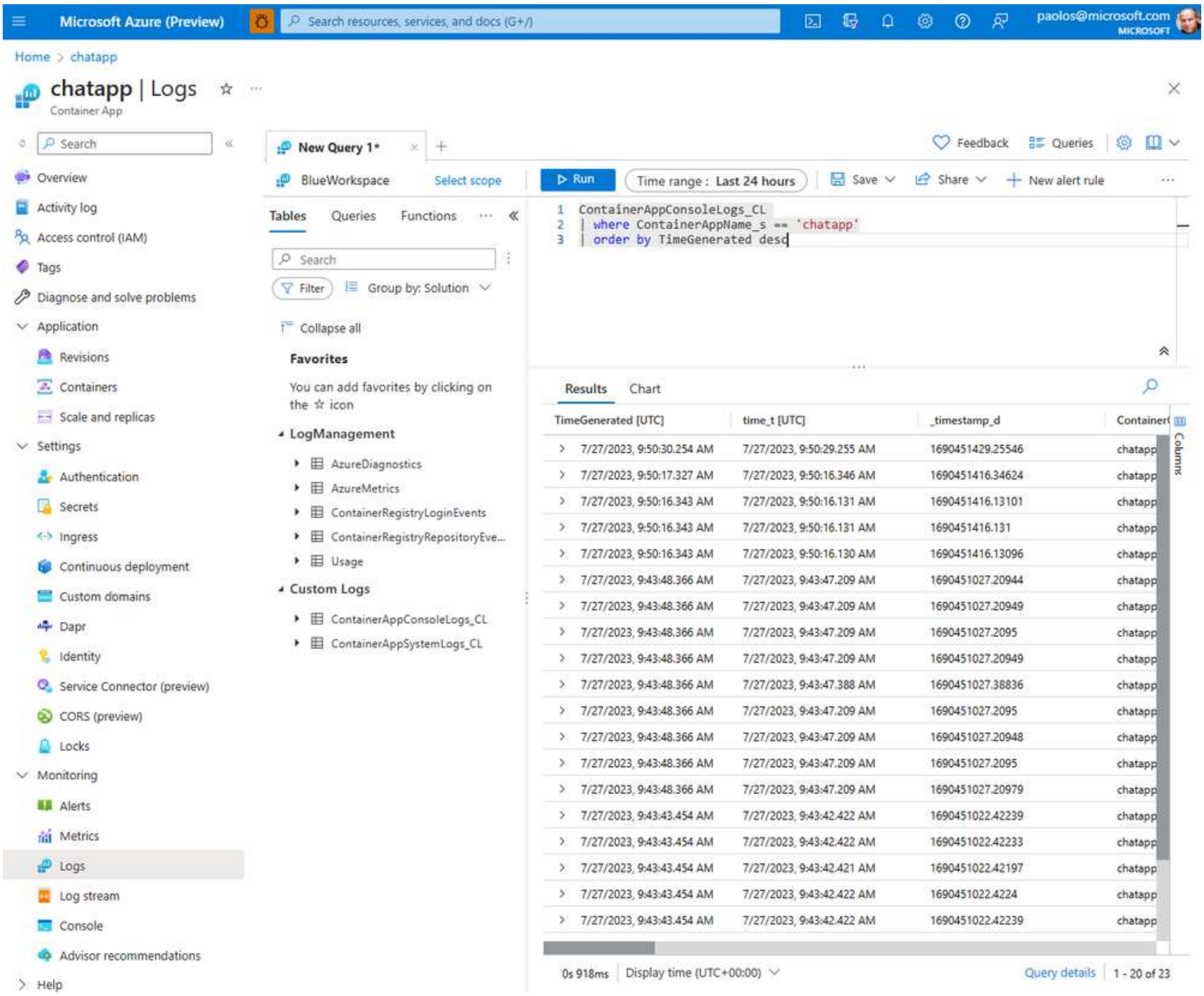
Logs ● Console ○ System

Replica: chatapp--39jy3n9-55db787494-zmt2d

Container: chat

```
g to the container 'chat'...
2023-07-27T09:43:30.77227  Successfully Connected to container: 'chat' [Revision: 'chatapp--39jy3n9', Replic
a: 'chatapp--39jy3n9-55db787494-zmt2d']
2023-07-26T23:35:16.142282374Z 2023-07-26 23:35:16 - Incomplete environment configuration. These variables a
re set: AZURE_CLIENT_ID
2023-07-26T23:35:16.142329430Z 2023-07-26 23:35:16 - ManagedIdentityCredential will use App Service managed
identity
2023-07-26T23:35:16.224265592Z 2023-07-26 23:35:16 - Your app is available at http://localhost:8000
2023-07-27T09:43:42.421974472Z 2023-07-27 09:43:42 - Request URL: 'http://localhost:42356/msi/token?api-vers
ion=REDACTED&resource=REDACTED&client_id=REDACTED'
2023-07-27T09:43:42.422329602Z Request method: 'GET'
2023-07-27T09:43:42.422385299Z Request headers:
2023-07-27T09:43:42.422392898Z     'X-IDENTITY-HEADER': 'REDACTED'
2023-07-27T09:43:42.422398668Z     'User-Agent': 'azsdk-python-identity/1.13.0 Python/3.11.4 (Linux-5.15.0-1
041-azure-x86_64-with-glibc2.36)'
2023-07-27T09:43:42.422403678Z No body was attached to the request
2023-07-27T09:43:47.209441675Z 2023-07-27 09:43:47 - Response status: 200
2023-07-27T09:43:47.209480291Z Response headers:
2023-07-27T09:43:47.209487616Z     'Content-Type': 'application/json; charset=utf-8'
2023-07-27T09:43:47.209491993Z     'Date': 'Thu, 27 Jul 2023 09:43:46 GMT'
2023-07-27T09:43:47.209496223Z     'Server': 'Kestrel'
2023-07-27T09:43:47.209500404Z     'Transfer-Encoding': 'chunked'
2023-07-27T09:43:47.209504184Z     'X-CORRELATION-ID': 'REDACTED'
2023-07-27T09:43:47.209790770Z 2023-07-27 09:43:47 - DefaultAzureCredential acquired a token from ManagedIde
ntityCredential
2023-07-27T09:43:47.388357476Z 2023-07-27 09:43:47 - message='OpenAI API response' path=https://blueopenai.o
penai.azure.com//openai/deployments/gpt-35-turbo-16k/chat/completions?api-version=2023-06-01-preview process
ing_ms=111.8376 request_id=7bee6697-a7a7-45a6-a782-a562a101f201 response_code=200
2023-07-27T09:44:48.27245  No logs since last 60 seconds
2023-07-27T09:45:48.68011  No logs since last 60 seconds
```

Alternatively, you can click open the `Logs` panel, as shown in the following screenshot, and use a Kusto Query Language (KQL) query to filter, project, and retrieve only the desired data.



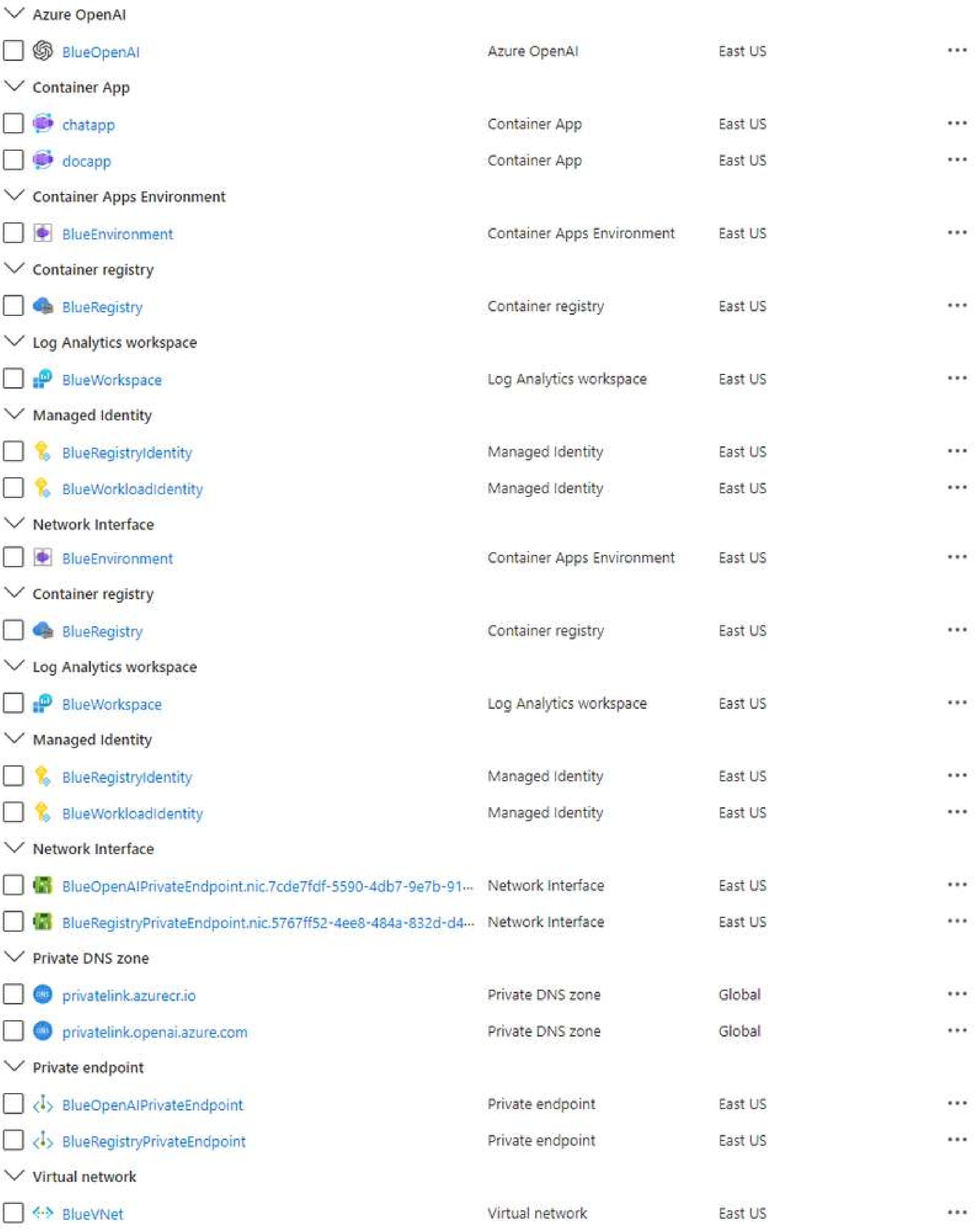
```
ContainerAppConsoleLogs_CL
| where ContainerAppName_s == 'chatapp'
| order by TimeGenerated desc
```

TimeGenerated [UTC]	time_t [UTC]	_timestamp_d	Container(
7/27/2023, 9:50:30.254 AM	7/27/2023, 9:50:29.255 AM	1690451429.25546	chatapp
7/27/2023, 9:50:17.327 AM	7/27/2023, 9:50:16.346 AM	1690451416.34624	chatapp
7/27/2023, 9:50:16.343 AM	7/27/2023, 9:50:16.131 AM	1690451416.13101	chatapp
7/27/2023, 9:50:16.343 AM	7/27/2023, 9:50:16.131 AM	1690451416.131	chatapp
7/27/2023, 9:50:16.343 AM	7/27/2023, 9:50:16.130 AM	1690451416.13096	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.20944	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.20949	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.2095	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.20949	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.388 AM	1690451027.38836	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.2095	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.20948	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.2095	chatapp
7/27/2023, 9:43:48.366 AM	7/27/2023, 9:43:47.209 AM	1690451027.20979	chatapp
7/27/2023, 9:43:43.454 AM	7/27/2023, 9:43:42.422 AM	1690451022.42239	chatapp
7/27/2023, 9:43:43.454 AM	7/27/2023, 9:43:42.422 AM	1690451022.42233	chatapp
7/27/2023, 9:43:43.454 AM	7/27/2023, 9:43:42.421 AM	1690451022.42197	chatapp
7/27/2023, 9:43:43.454 AM	7/27/2023, 9:43:42.422 AM	1690451022.4224	chatapp
7/27/2023, 9:43:43.454 AM	7/27/2023, 9:43:42.422 AM	1690451022.42239	chatapp

0s 918ms | Display time (UTC+00:00) | Query details | 1 - 20 of 23

Review deployed resources

You can use the Azure portal to list the deployed resources in the resource group, as shown in the following picture:



∨	Azure OpenAI			
□	BlueOpenAI	Azure OpenAI	East US	•••
∨	Container App			
□	chatapp	Container App	East US	•••
□	docapp	Container App	East US	•••
∨	Container Apps Environment			
□	BlueEnvironment	Container Apps Environment	East US	•••
∨	Container registry			
□	BlueRegistry	Container registry	East US	•••
∨	Log Analytics workspace			
□	BlueWorkspace	Log Analytics workspace	East US	•••
∨	Managed Identity			
□	BlueRegistryIdentity	Managed Identity	East US	•••
□	BlueWorkloadIdentity	Managed Identity	East US	•••
∨	Network Interface			
□	BlueEnvironment	Container Apps Environment	East US	•••
∨	Container registry			
□	BlueRegistry	Container registry	East US	•••
∨	Log Analytics workspace			
□	BlueWorkspace	Log Analytics workspace	East US	•••
∨	Managed Identity			
□	BlueRegistryIdentity	Managed Identity	East US	•••
□	BlueWorkloadIdentity	Managed Identity	East US	•••
∨	Network Interface			
□	BlueOpenAIPrivateEndpoint.nic.7cde7fdf-5590-4db7-9e7b-91...	Network Interface	East US	•••
□	BlueRegistryPrivateEndpoint.nic.5767ff52-4ee8-484a-832d-d4...	Network Interface	East US	•••
∨	Private DNS zone			
□	privatelink.azurecr.io	Private DNS zone	Global	•••
□	privatelink.openai.azure.com	Private DNS zone	Global	•••
∨	Private endpoint			
□	BlueOpenAIPrivateEndpoint	Private endpoint	East US	•••
□	BlueRegistryPrivateEndpoint	Private endpoint	East US	•••
∨	Virtual network			
□	BlueVNet	Virtual network	East US	•••

You can also use Azure CLI to list the deployed resources in the resource group:

```
az resource list --resource-group <resource-group-name>
```

You can also use the following PowerShell cmdlet to list the deployed resources in the resource group:

```
Get-AzResource -ResourceGroupName <resource-group-name>
```

Skip to main content
Skip to primary navigation

Clean up resources

You can delete the resource group using the following Azure CLI command when you no longer need the resources you created. This will remove all the Azure resources.

```
az group delete --name <resource-group-name>
```

Alternatively, you can use the following PowerShell cmdlet to delete the resource group and all the Azure resources.

```
Remove-AzResourceGroup -Name <resource-group-name>
```

3 Likes

6 Comments

Jul 27 2023 08:40 A

cicorias Microsoft

Thanks again for such relevant and timely guidance Paolo!!

1 Like

Jul 27 2023 09:01 A

paolosalvatori Microsoft

Thanks @cicorias, I thought to create an Azure Container Apps + Azure OpenAI sample after my articles on AKS + Azure OpenAI.

1 Like

Aug 15 2023 02:32

merveguel Copper Contributor

Hi @paolosalvatori, thanks for such detailed post, I was wondering if this article can be implemented using the free-tier version of the prerequisite tools?
Best, Merve

0 Likes

Aug 15 2023 03:17

paolosalvatori Microsoft

Skip to primary navigation Skip to main content

Hi @merveguel, all the pre-requisites are free of charge. I think you can use a free Azure account to test the entire architecture, but I'm not sure about Azure OpenAI Service. I should check, but I'm OOF now. If you don't have an Azure account already, the best way is to open one. Alternatively, check the pricing page for the various services and the Azure Free Account FAQ: https://azure.microsoft.com/en-us/free/free-account-faq

1 Like

Aug 29 2023 01:02

Heman_k Copper Contributor

Hi @paolosalvatori , Thanks for sharing this very detailed article. Covers a lot of ground!!

On a related topic, can you please comment on the vector search capabilities in Azure Search? How does it compare it with vector search in other similar products?

Also, I am specifically interested in the support for HNSW algorithm in in Azure Search. Does Azure Search support HNSW? I have seen sample code where there is a module called HnswVectorSearchAlgorithmConfiguration in the azure.search.documents.indexes.models package, but I am having problems importing this module, at least in Python 3.9. Any info or suggestions?

0 Likes

Aug 29 2023 11:59

paolosalvatori Microsoft

Thanks @Heman_k

I'm not an Azure Cognitive Search subject-matter expert, but I can surely affirm that Azure Cognitive Search has strong vector search capabilities that allow you to search and retrieve similar vectors based on their similarity scores. While I do not have direct comparisons to other similar products or other vector databases, such as Chroma or FAISS, Azure Cognitive Search's vector search capabilities are designed to provide efficient and accurate results.
Regarding the HNSW algorithm, Azure Search does support it. The HNSW algorithm is a commonly used algorithm for approximate nearest neighbor search, and it is available in Azure Cognitive Search for vector search scenarios.

Skip to primary navigation
Skip to main content

Please check Add vector search - Azure Cognitive Search | Microsoft Learn.
As for the specific issue you mentioned regarding importing the HnswVectorSearchAlgorithmConfiguration module in Python 3.9, it's possible that the module may not be available or accessible in that specific version. It is recommended to check the documentation or reach out to Microsoft Azure support for further assistance or alternative approaches.
Overall, Azure Search offers robust vector search capabilities, including support for the HNSW algorithm, which can be valuable for creating efficient and accurate search experiences.

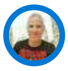
P.S. If you found my article and sample interesting and helpful, please like the article and star the GitHub project, thanks :)

1 Like

You must be a registered user to add a comment. If you've already registered, sign in. Otherwise, register and sign in.

Comment

Co-Authors



paolosalvatori

Version history

Last update: Jul 27 2023 07:40 AM
Updated by: paolosalvatori

Labels

App	101
Cloud Native Apps	70
Data & AI	110

Skip to primary navigation
Skip to main content

Infra	89
ISV 1:Many	37

‹ Previous   Next ›

Share

What's new

Surface Pro 9

Surface Laptop 5

Surface Studio 2+

Surface Laptop Go 2

Surface Laptop Studio

Surface Duo 2

Microsoft 365

Windows 11 apps

Microsoft Store

Account profile

Download Center

Microsoft Store support

Returns

Order tracking

Virtual workshops and training

Microsoft Store Promise

Flexible Payments

Education

Microsoft in education

Devices for education

Microsoft Teams for Education

Skip to footer navigation

Skip to main content

Microsoft 365 Education

Education consultation appointment

Educator training and development

Deals for students and parents

Azure for students

Business

Microsoft Cloud

Microsoft Security

Dynamics 365

Microsoft 365

Microsoft Power Platform

Microsoft Teams

Microsoft Industry

Small Business

Developer & IT

Azure

Developer Center

Documentation

Microsoft Learn

Microsoft Tech Community

Azure Marketplace

AppSource

Visual Studio

Company

Careers

About Microsoft

Company news

Privacy at Microsoft

Investors

Diversity and inclusion

Accessibility

Skip to primary navigation

Skip to main content

Sustainability