Contents

What is Computer Vision?

6/22/2021 • 2 minutes to read • Edit Online

IMPORTANT

Transport Layer Security (TLS) 1.2 is now enforced for all HTTP requests to this service. For more information, see Azure Cognitive Services security.

Azure's Computer Vision service gives you access to advanced algorithms that process images and return information based on the visual features you're interested in.

SERVICE	DESCRIPTION
Optical Character Recognition (OCR)	The Optical Character Recognition (OCR) service extracts text from images. You can use the new Read API to extract printed and handwritten text from photos and documents. It uses deep-learning-based models and works with text on a variety of surfaces and backgrounds. These include business documents, invoices, receipts, posters, business cards, letters, and whiteboards. The OCR APIs support extracting printed text in several languages. Follow the OCR quickstart to get started.
Image Analysis	The Image Analysis service extracts many visual features from images, such as objects, faces, adult content, and auto-generated text descriptions. Follow the Image Analysis quickstart to get started.
Spatial Analysis	The Spatial Analysis service analyzes the presence and movement of people on a video feed and produces events that other systems can respond to. Install the Spatial Analysis container to get started.

Computer Vision for digital asset management

Computer Vision can power many digital asset management (DAM) scenarios. DAM is the business process of organizing, storing, and retrieving rich media assets and managing digital rights and permissions. For example, a company may want to group and identify images based on visible logos, faces, objects, colors, and so on. Or, you might want to automatically generate captions for images and attach keywords so they're searchable. For an all-in-one DAM solution using Cognitive Services, Azure Cognitive Search, and intelligent reporting, see the Knowledge Mining Solution Accelerator Guide on GitHub. For other DAM examples, see the Computer Vision Solution Templates repository.

Image requirements

Computer Vision can analyze images that meet the following requirements:

- The image must be presented in JPEG, PNG, GIF, or BMP format
- The file size of the image must be less than 4 megabytes (MB)
- The dimensions of the image must be greater than 50 x 50 pixels
  - For the Read API, the dimensions of the image must be between 50 x 50 and 10000 x 10000 pixels.

Data privacy and security

As with all of the Cognitive Services, developers using the Computer Vision service should be aware of Microsoft's policies on customer data. See the Cognitive Services page on the Microsoft Trust Center to learn more.

Next steps

Follow a quickstart to implement and run a service in your preferred development language.

- Quickstart: Optical character recognition (OCR)
- Quickstart: Image Analysis
- Quickstart: Spatial Analysis container

Language support for Computer Vision

6/17/2021 • 2 minutes to read • Edit Online

Some features of Computer Vision support multiple languages; any features not mentioned here only support English.

Optical Character Recognition (OCR)

Computer Vision's OCR APIs support several languages. They do not require you to specify a language code. See the Optical Character Recognition (OCR) overview for more information.

LANGUAGE	LANGUAGE CODE	READ 3.2	OCR API	READ 3.0/3.1
Afrikaans	`af`	✔		
Albanian	`sq`	✔		
Arabic	`ar`		✔	
Asturian	`ast`	✔		
Basque	`eu`	✔		
Bislama	`bi`	✔		
Breton	`br`	✔		
Catalan	`ca`	✔		
Cebuano	`ceb`	✔		
Chamorro	`ch`	✔		
Chinese Simplified	`zh-Hans`	✔	✔	
Chinese Traditional	`zh-Hant`	✔	✔	
Cornish	`kw`	✔		
Corsican	`co`	✔		
Crimean Tatar Latin	`crh`	✔		
Czech	`cs`	✔	✔	
Danish	`da`	✔	✔	

LANGUAGE	LANGUAGE CODE	READ 3.2	OCR API	READ 3.0/3.1
Dutch	`nl`	✔	✔	✔
English (incl. handwritten)	`en`	✔	✔ (print only)	✔
Estonian	`et`	✔		
Fijian	`fj`	✔		
Filipino	`fil`	✔		
Finnish	`fi`	✔	✔	
French	`fr`	✔	✔	✔
Friulian	`fur`	✔		
Galician	`gl`	✔		
German	`de`	✔	✔	✔
Gilbertese	`gil`	✔		
Greek	`el`		✔	
Greenlandic	`kl`	✔		
Haitian Creole	`ht`	✔		
Hani	`hni`	✔		
Hmong Daw Latin	`mww`	✔		
Hungarian	`hu`	✔	✔	
Indonesian	`id`	✔		
Interlingua	`ia`	✔		
Inuktitut Latin	`iu`	✔		
Irish	`ga`	✔		
Italian	`it`	✔	✔	✔
Japanese	`ja`	✔	✔	
Javanese	`jv`	✔		

LANGUAGE	LANGUAGE CODE	READ 3.2	OCR API	READ 3.0/3.1
K'iche'	`quc`	✔		
Kabuverdianu	`kea`	✔		
Kachin Latin	`kac`	✔		
Kara-Kalpak	`kaa`	✔		
Kashubian	`csb`	✔		
Khasi	`kha`	✔		
Korean	`ko`	✔	✔	
Kurdish Latin	`kur`	✔		
Luxembourgish	`lb`	✔		
Malay Latin	`ms`	✔		
Manx	`gv`	✔		
Neapolitan	`nap`	✔		
Norwegian	`nb`		✔	
Norwegian	`no`	✔		
Occitan	`oc`	✔		
Polish	`pl`	✔	✔	
Portuguese	`pt`	✔	✔	✔
Romanian	`ro`		✔	
Romansh	`rm`	✔		
Russian	`ru`		✔	
Scots	`sco`	✔		
Scottish Gaelic	`gd`	✔		
Serbian Cyrillic	`sr-Cyrl`		✔	
Serbian Latin	`sr-Latn`		✔	

LANGUAGE	LANGUAGE CODE	READ 3.2	OCR API	READ 3.0/3.1
Slovak	`sk`		✔	
Slovenian	`slv`	✔		
Spanish	`es`	✔	✔	✔
Swahili Latin	`sw`	✔		
Swedish	`sv`	✔	✔	
Tatar Latin	`tat`	✔		
Tetum	`tet`	✔		
Turkish	`tr`	✔	✔	
Upper Sorbian	`hsb`	✔		
Uzbek Latin	`uz`	✔		
Volapük	`vo`	✔		
Walser	`wae`	✔		
Western Frisian	`fy`	✔		
Yucatec Maya	`yua`	✔		
Zhuang	`za`	✔		
Zulu	`zu`	✔		

Image analysis

Some actions of the Analyze - Image API can return results in other languages, specified with the `language` query parameter. Other actions return results in English regardless of what language is specified, and others throw an exception for unsupported languages. Actions are specified with the `visualFeatures` and `details` query parameters; see the Overview for a list of all the actions you can do with image analysis.

LANGUAGE	LANGUAGE CODE	CATEGORIES	TAGS	DESCRIPTION	ADULT	BRANDS	COLOR	FACES	IMAGETYPE	OBJECTS	CELEBRITIES	LANDMARKS
Chinese	`zh`	✔	✔	✔	-	-	-	-	-	✘	✔	✔
English	`en`	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

LANGUAGE	LANGUAGE CODE	CATEGORIES	TAGS	DESCRIPTION	ADULT	BRANDS	COLOR	FACES	IMAGETYPE	OBJECTS	CELEBRITIES	LANDMARKS
Japanese	`ja`	✔	✔	✔	-	-	-	-	-	✘	✔	✔
Portuguese	`pt`	✔	✔	✔	-	-	-	-	-	✘	✔	✔
Spanish	`es`	✔	✔	✔	-	-	-	-	-	✘	✔	✔

What's new in Computer Vision

5/25/2021 • 5 minutes to read • Edit Online

Learn what's new in the service. These items may be release notes, videos, blog posts, and other types of information. Bookmark this page to stay up to date with the service.

May 2021

Spatial Analysis container update

A new version of the Spatial Analysis container has been released with a new feature set. This Docker container lets you analyze real-time streaming video to understand spatial relationships between people and their movement through physical environments.

- Spatial Analysis operations can be now configured to detect the orientation that a person is facing.
  - An orientation classifier can be enabled for the `personcrossingline` and `personcrossingpolygon` operations by configuring the `enable_orientation` parameter. It is set to off by default.
- Spatial Analysis operations now also offers configuration to detect a person's speed while walking/running
  - Speed can be detected for the `personcrossingline` and `personcrossingpolygon` operations by turning on the `enable_speed` classifier, which is off by default. The output is reflected in the `speed`, `avgSpeed`, and `minSpeed` outputs.

April 2021

Computer Vision v3.2 GA

The Computer Vision API v3.2 is now generally available with the following updates:

- Improved image tagging model: analyzes visual content and generates relevant tags based on objects, actions, and content displayed in the image. This model is available through the Tag Image API. See the Image Analysis how-to guide and overview to learn more.
- Updated content moderation model: detects presence of adult content and provides flags to filter images containing adult, racy, and gory visual content. This model is available through the Analyze API. See the Image Analysis how-to guide and overview to learn more.
- OCR (Read) available for 73 languages including Simplified and Traditional Chinese, Japanese, Korean, and Latin languages.
- OCR (Read) also available as a Distroless container for on-premise deployment.

See Computer Vision v3.2 GA

March 2021

Computer Vision 3.2 Public Preview update

The Computer Vision API v3.2 public preview has been updated. The preview release has all Computer Vision features along with updated Read and Analyze APIs.

See Computer Vision v3.2 public preview 3

February 2021

Read API v3.2 Public Preview with OCR support for 73 languages

Computer Vision's Read API v3.2 public preview, available as cloud service and Docker container, includes these updates:

- OCR for 73 languages including Simplified and Traditional Chinese, Japanese, Korean, and Latin languages.
- Natural reading order for the text line output (Latin languages only)
- Handwriting style classification for text lines along with a confidence score (Latin languages only).
- Extract text only for selected pages for a multi-page document.
- Available as a Distroless container for on-premise deployment.

See the Read API how-to guide to learn more.

Use the Read API v3.2 Public Preview

January 2021

Spatial Analysis container update

A new version of the Spatial Analysis container has been released with a new feature set. This Docker container lets you analyze real-time streaming video to understand spatial relationships between people and their movement through physical environments.

- Spatial Analysis operations can be now configured to detect if a person is wearing a protective face covering such as a mask.
  - A mask classifier can be enabled for the `personcount`, `personcrossingline` and `personcrossingpolygon` operations by configuring the `ENABLE_FACE_MASK_CLASSIFIER` parameter.
  - The attributes `face_mask` and `face_noMask` will be returned as metadata with confidence score for each person detected in the video stream
- The *personcrossingpolygon* operation has been extended to allow the calculation of the dwell time a person spends in a zone. You can set the `type` parameter in the Zone configuration for the operation to `zonedwelltime` and a new event of type *personZoneDwellTimeEvent* will include the `durationMs` field populated with the number of milliseconds that the person spent in the zone.
- **Breaking change**: The *personZoneEvent* event has been renamed to *personZoneEnterExitEvent*. This event is raised by the *personcrossingpolygon* operation when a person enters or exits the zone and provides directional info with the numbered side of the zone that was crossed.
- Video URL can be provided as "Private Parameter/obfuscated" in all operations. Obfuscation is optional now and it will only work if `KEY` and `IV` are provided as environment variables.
- Calibration is enabled by default for all operations. Set the `do_calibration: false` to disable it.
- Added support for auto recalibration (by default disabled) via the `enable_recalibration` parameter, please refer to Spatial Analysis operations for details
- Camera calibration parameters to the `DETECTOR_NODE_CONFIG`. Refer to Spatial Analysis operations for details.

October 2020

Computer Vision API v3.1 GA

The Computer Vision API in General Availability has been upgraded to v3.1.

September 2020

Spatial Analysis container preview

The Spatial Analysis container is now in preview. The Spatial Analysis feature of Computer Vision lets you analyze real-time streaming video to understand spatial relationships between people and their movement

through physical environments. Spatial Analysis is a Docker container you can use on-premises.

Read API v3.1 Public Preview adds OCR for Japanese

Computer Vision's Read API v3.1 public preview adds these capabilities:

- OCR for Japanese language
- For each text line, indicate whether the appearance is Handwriting or Print style, along with a confidence score (Latin languages only).
- For a multi-page document extract text only for selected pages or page range.
- This preview version of the Read API supports English, Dutch, French, German, Italian, Japanese, Portuguese, Simplified Chinese, and Spanish languages.

See the Read API how-to guide to learn more.

Learn more about Read API v3.1 Public Preview 2

July 2020

Read API v3.1 Public Preview with OCR for Simplified Chinese

Computer Vision's Read API v3.1 public preview adds support for Simplified Chinese.

- This preview version of the Read API supports English, Dutch, French, German, Italian, Portuguese, Simplified Chinese, and Spanish languages.

See the Read API how-to guide to learn more.

Learn more about Read API v3.1 Public Preview 1

May 2020

Computer Vision API v3.0 entered General Availability, with updates to the Read API:

- Support for English, Dutch, French, German, Italian, Portuguese, and Spanish
- Improved accuracy
- Confidence score for each extracted word
- New output format

See the OCR overview to learn more.

March 2020

- TLS 1.2 is now enforced for all HTTP requests to this service. For more information, see Azure Cognitive Services security.

January 2020

Read API 3.0 Public Preview

You now can use version 3.0 of the Read API to extract printed or handwritten text from images. Compared to earlier versions, 3.0 provides:

- Improved accuracy
- New output format
- Confidence score for each extracted word
- Support for both Spanish and English languages with the language parameter

Follow an Extract text quickstart to get starting using the 3.0 API.

Cognitive Service updates

Azure update announcements for Cognitive Services

Computer Vision API Frequently Asked Questions

6/10/2021 • 2 minutes to read • Edit Online

> **TIP**
> If you can't find answers to your questions in this FAQ, try asking the Computer Vision API community on StackOverflow or contact Help and Support on UserVoice

Question: *Can I train Computer Vision API to use custom tags? For example, I would like to feed in pictures of cat breeds to 'train' the AI, then receive the breed value on an AI request.*

Answer: This function is currently not available. However, our engineers are working to bring this functionality to Computer Vision.

Question: *Can I deploy the OCR (Read) capability on-premise?*

Answer: Yes, the OCR (Read) cloud API is also available as a Docker container for on-premise deployment. Learn how to deploy the OCR containers.

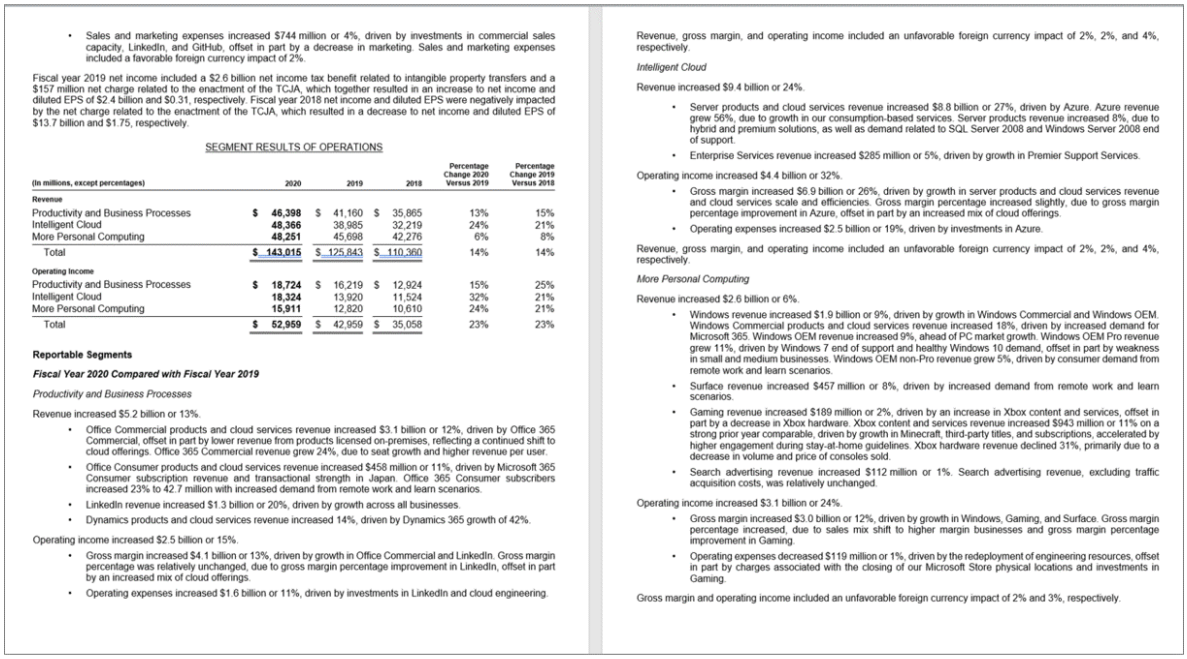
Question: *Can Computer Vision be used to read license plates?*

Answer: The Vision API includes the deep learning powered OCR capabilities with the latest Read feature. We are constantly trying to improve our services to work across all scenarios.

What is Optical character recognition?

6/22/2021 • 2 minutes to read • Edit Online

Optical character recognition (OCR) allows you to extract printed or handwritten text from images, such as photos of street signs and products, as well as from documents—invoices, bills, financial reports, articles, and more. Microsoft's OCR technologies support extracting printed text in several languages. Follow a quickstart to get started.



- Sales and marketing expenses increased $744 million or 4%, driven by investments in commercial sales capacity, LinkedIn, and GitHub, offset in part by a decrease in marketing. Sales and marketing expenses included a favorable foreign currency impact of 2%.

Fiscal year 2019 net income included a $2.6 billion net income tax benefit related to intangible property transfers and a $157 million net charge related to the enactment of the TCJA, which together resulted in an increase to net income and diluted EPS of $2.4 billion and $0.31, respectively. Fiscal year 2018 net income and diluted EPS were negatively impacted by the net charge related to the enactment of the TCJA, which resulted in a decrease to net income and diluted EPS of $13.7 billion and $1.75, respectively.

SEGMENT RESULTS OF OPERATIONS

(In millions, except percentages)	2020	2019	2018	Percentage Change 2020 Versus 2019	Percentage Change 2019 Versus 2018
Revenue					
Productivity and Business Processes	$ 46,398	$ 41,160	$ 35,865	13%	15%
Intelligent Cloud	48,366	38,985	32,219	24%	21%
More Personal Computing	48,251	45,698	42,276	6%	8%
Total	$ 143,015	$ 125,843	$ 110,360	14%	14%
Operating Income					
Productivity and Business Processes	$ 18,724	$ 16,219	$ 12,924	15%	25%
Intelligent Cloud	18,324	13,920	11,524	32%	21%
More Personal Computing	15,911	12,820	10,610	24%	21%
Total	$ 52,959	$ 42,959	$ 35,058	23%	23%

Reportable Segments

Fiscal Year 2020 Compared with Fiscal Year 2019

Productivity and Business Processes

Revenue increased $5.2 billion or 13%.

- Office Commercial products and cloud services revenue increased $3.1 billion or 12%, driven by Office 365 Commercial, offset in part by lower revenue from products licensed on-premises, reflecting a continued shift to cloud offerings. Office 365 Commercial revenue grew 24%, due to seat growth and higher revenue per user.
- Office Consumer products and cloud services revenue increased $458 million or 11%, driven by Microsoft 365 Consumer subscription revenue and transactional strength in Japan. Office 365 Consumer subscribers increased 23% to 42.7 million with increased demand from remote work and learn scenarios.
- LinkedIn revenue increased $1.3 billion or 20%, driven by growth across all businesses.
- Dynamics products and cloud services revenue increased 14%, driven by Dynamics 365 growth of 42%.

Operating income increased $2.5 billion or 15%.

- Gross margin increased $4.1 billion or 13%, driven by growth in Office Commercial and LinkedIn. Gross margin percentage was relatively unchanged, due to gross margin percentage improvement in LinkedIn, offset in part by an increased mix of cloud offerings.
- Operating expenses increased $1.6 billion or 11%, driven by investments in LinkedIn and cloud engineering.

Revenue, gross margin, and operating income included an unfavorable foreign currency impact of 2%, 2%, and 4%, respectively.

Intelligent Cloud

Revenue increased $9.4 billion or 24%.

- Server products and cloud services revenue increased $8.8 billion or 27%, driven by Azure. Azure revenue grew 56%, due to growth in our consumption-based services. Server products revenue increased 8%, due to hybrid and premium solutions, as well as demand related to SQL Server 2008 and Windows Server 2008 end of support.
- Enterprise Services revenue increased $285 million or 5%, driven by growth in Premier Support Services.

Operating income increased $4.4 billion or 32%.

- Gross margin increased $6.9 billion or 26%, driven by growth in server products and cloud services revenue and cloud services scale and efficiencies. Gross margin percentage increased slightly, due to gross margin percentage improvement in Azure, offset in part by an increased mix of cloud offerings.
- Operating expenses increased $2.5 billion or 19%, driven by investments in Azure.

Revenue, gross margin, and operating income included an unfavorable foreign currency impact of 2%, 2%, and 4%, respectively.

More Personal Computing

Revenue increased $2.6 billion or 6%.

- Windows revenue increased $1.9 billion or 9%, driven by growth in Windows Commercial and Windows OEM. Windows Commercial products and cloud services revenue increased 18%, driven by increased demand for Microsoft 365. Windows OEM revenue increased 9%, ahead of PC market growth. Windows OEM Pro revenue grew 11%, driven by Windows 7 end of support and healthy Windows 10 demand, offset in part by weakness in small and medium businesses. Windows OEM non-Pro revenue grew 5%, driven by consumer demand from remote work and learn scenarios.
- Surface revenue increased $457 million or 8%, driven by increased demand from remote work and learn scenarios.
- Gaming revenue increased $189 million or 2%, driven by an increase in Xbox content and services, offset in part by a decrease in Xbox hardware. Xbox content and services revenue increased $943 million or 11% on a strong prior year comparable, driven by growth in Minecraft, third-party titles, and subscriptions, accelerated by higher engagement during stay-at-home guidelines. Xbox hardware revenue declined 31%, primarily due to a decrease in volume and price of consoles sold.
- Search advertising revenue increased $112 million or 1%. Search advertising revenue, excluding traffic acquisition costs, was relatively unchanged.

Operating income increased $3.1 billion or 24%.

- Gross margin increased $3.0 billion or 12%, driven by growth in Windows, Gaming, and Surface. Gross margin percentage increased, due to sales mix shift to higher margin businesses and gross margin percentage improvement in Gaming.
- Operating expenses decreased $119 million or 1%, driven by the redeployment of engineering resources, offset in part by charges associated with the closing of our Microsoft Store physical locations and investments in Gaming.

Gross margin and operating income included an unfavorable foreign currency impact of 2% and 3%, respectively.

This documentation contains the following types of articles:

- The quickstarts are step-by-step instructions that let you make calls to the service and get results in a short period of time.
- The how-to guides contain instructions for using the service in more specific or customized ways.

Read API

The Computer Vision Read API is Azure's latest OCR technology (learn what's new) that extracts printed text (in several languages), handwritten text (English only), digits, and currency symbols from images and multi-page PDF documents. It's optimized to extract text from text-heavy images and multi-page PDF documents with mixed languages. It supports detecting both printed and handwritten text in the same image or document.

Input requirements

The **Read** call takes images and documents as its input. They have the following requirements:

- Supported file formats: JPEG, PNG, BMP, PDF, and TIFF
- For PDF and TIFF files, up to 2000 pages (only first two pages for the free tier) are processed.
- The file size must be less than 50 MB (6 MB for the free tier) and dimensions at least 50 x 50 pixels and at most 10000 x 10000 pixels.

Supported languages

The Read API supports a total of 73 languages for print style text. Refer to the full list of OCR-supported languages. Handwritten-style OCR is supported exclusively for English.

Key features

The Read API includes the following features.

- Print text extraction in 73 languages
- Handwritten text extraction in English
- Text lines and words with location and confidence scores
- No language identification required
- Support for mixed languages, mixed mode (print and handwritten)
- Select pages and page ranges from large, multi-page documents
- Natural reading order for text lines
- Handwriting classification for text lines
- Available as Distroless Docker container for on-premise deployment

Learn how to use the OCR features.

Use the cloud API or deploy on-premise

The Read 3.x cloud APIs are the preferred option for most customers because of ease of integration and fast productivity out of the box. Azure and the Computer Vision service handle scale, performance, data security, and compliance needs while you focus on meeting your customers' needs.

For on-premise deployment, the Read Docker container (preview) enables you to deploy the new OCR capabilities in your own local environment. Containers are great for specific security and data governance requirements.

> **WARNING**
> The Computer Vision 2.0 RecognizeText operations are in the process of being deprecated in favor of the new Read API covered in this article. Existing customers should transition to using Read operations.

Data privacy and security

As with all of the Cognitive Services, developers using the Computer Vision service should be aware of Microsoft's policies on customer data. See the Cognitive Services page on the Microsoft Trust Center to learn more.

Next steps

- Get started with the OCR (Read) REST API or client library quickstarts.
- Learn about the Read 3.2 REST API.

Quickstart: Use the Read client library or REST API

4/20/2021 • 33 minutes to read • Edit Online

Get started with the Read REST API or client libraries. The Read service provides you with AI algorithms for extracting visible text from images and returning it as structured strings. Follow these steps to install a package to your application and try out the sample code for basic tasks.

Use the OCR client library to read printed and handwritten text from an image.

Reference documentation | Library source code | Package (NuGet) | Samples

Prerequisites

- An Azure subscription - Create one for free
- The Visual Studio IDE or current version of .NET Core.
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a new C# application

- Visual Studio IDE
- CLI

Using Visual Studio, create a new .NET Core application.

Install the client library

Once you've created a new project, install the client library by right-clicking on the project solution in the **Solution Explorer** and selecting **Manage NuGet Packages**. In the package manager that opens select **Browse**, check **Include prerelease**, and search for `Microsoft.Azure.CognitiveServices.Vision.ComputerVision`. Select version `7.0.0`, and then **Install**.

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

From the project directory, open the *Program.cs* file in your preferred editor or IDE.

Find the subscription key and endpoint

Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your subscription key and endpoint in the resource's **key and endpoint** page, under **resource management**.

In the application's **Program** class, create variables for your Computer Vision subscription key and endpoint.

Paste your subscription key and endpoint into the following code where indicated. Your Computer Vision endpoint has the form `https://<your_computer_vision_resource_name>.cognitiveservices.azure.com/`.

```
using System;
using System.Collections.Generic;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
using System.Threading.Tasks;
using System.IO;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System.Threading;
using System.Linq;

namespace ComputerVisionQuickstart
{
    class Program
    {
        // Add your Computer Vision subscription key and endpoint
        static string subscriptionKey = "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE";
        static string endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE";
```

IMPORTANT

Remember to remove the subscription key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. For example, Azure key vault.

In the application's `Main` method, add calls for the methods used in this quickstart. You will create these later.

```
ComputerVisionClient client = Authenticate(endpoint, subscriptionKey);

// Extract text (OCR) from a URL image using the Read API
ReadFileUrl(client, READ_TEXT_URL_IMAGE).Wait();
```

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the OCR .NET SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to do most image operations.
ComputerVisionClientExtensions	This class contains additional methods for the **ComputerVisionClient**.

Code examples

These code snippets show you how to do the following tasks with the OCR client library for .NET:

- Authenticate the client
- Read printed and handwritten text

Authenticate the client

In a new method in the **Program** class, instantiate a client with your endpoint and subscription key. Create a **ApiKeyServiceClientCredentials** object with your subscription key, and use it with your endpoint to create a **ComputerVisionClient** object.

```
/*
 * AUTHENTICATE
 * Creates a Computer Vision client used by each example.
 */
public static ComputerVisionClient Authenticate(string endpoint, string key)
{
    ComputerVisionClient client =
      new ComputerVisionClient(new ApiKeyServiceClientCredentials(key))
      { Endpoint = endpoint };
    return client;
}
```

I authenticated the client I ran into an issue

Read printed and handwritten text

The OCR service can read visible text in an image and convert it to a character stream. For more information on text recognition, see the Optical character recognition (OCR) overview. The code in this section uses the latest Computer Vision SDK release for Read 3.0 and defines a method, `BatchReadFileUrl`, which uses the client object to detect and extract text in the image.

> **TIP**
> You can also extract text from a local image. See the ComputerVisionClient methods, such as **ReadInStreamAsync**. Or, see the sample code on GitHub for scenarios involving local images.

Set up test image

In your **Program** class, save a reference to the URL of the image you want to extract text from. This snippet includes sample images for both printed and handwritten text.

```
private const string READ_TEXT_URL_IMAGE =
"https://intelligentkioskstore.blob.core.windows.net/visionapi/suggestedphotos/3.png";
```

Call the Read API

Define the new method for reading text. Add the code below, which calls the **ReadAsync** method for the given image. This returns an operation ID and starts an asynchronous process to read the content of the image.

```
/*
 * READ FILE - URL
 * Extracts text.
 */
public static async Task ReadFileUrl(ComputerVisionClient client, string urlFile)
{
    Console.WriteLine("----------------------------------------------------------");
    Console.WriteLine("READ FILE FROM URL");
    Console.WriteLine();

    // Read text from URL
    var textHeaders = await client.ReadAsync(urlFile);
    // After the request, get the operation location (operation ID)
    string operationLocation = textHeaders.OperationLocation;
    Thread.Sleep(2000);
```

Get Read results

Next, get the operation ID returned from the **ReadAsync** call, and use it to query the service for operation results. The following code checks the operation until the results are returned. It then prints the extracted text data to the console.

```
// Retrieve the URI where the extracted text will be stored from the Operation-Location header.
// We only need the ID and not the full URL
const int numberOfCharsInOperationId = 36;
string operationId = operationLocation.Substring(operationLocation.Length - numberOfCharsInOperationId);

// Extract the text
ReadOperationResult results;
Console.WriteLine($"Extracting text from URL file {Path.GetFileName(urlFile)}...");
Console.WriteLine();
do
{
    results = await client.GetReadResultAsync(Guid.Parse(operationId));
}
while ((results.Status == OperationStatusCodes.Running ||
    results.Status == OperationStatusCodes.NotStarted));
```

Display Read results

Add the following code to parse and display the retrieved text data, and finish the method definition.

```
    // Display the found text.
    Console.WriteLine();
    var textUrlFileResults = results.AnalyzeResult.ReadResults;
    foreach (ReadResult page in textUrlFileResults)
    {
        foreach (Line line in page.Lines)
        {
            Console.WriteLine(line.Text);
        }
    }
    Console.WriteLine();
}
```

Run the application

- Visual Studio IDE
- CLI

Run the application by clicking the **Debug** button at the top of the IDE window.

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

Next steps

In this quickstart, you learned how to install the OCR client library and use the Read API. Next, learn more about the Read API features.

Call the Read API

- OCR overview
- The source code for this sample can be found on GitHub.

Use the Optical character recognition client library to read printed and handwritten text with the Read API.

Reference documentation | Library source code | Package (PiPy) | Samples

Prerequisites

- An Azure subscription - Create one for free
- Python 3.x
  - Your Python installation should include pip. You can check if you have pip installed by running `pip --version` on the command line. Get pip by installing the latest version of Python.
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Install the client library

You can install the client library with:

```
pip install --upgrade azure-cognitiveservices-vision-computervision
```

Also install the Pillow library.

```
pip install pillow
```

Create a new Python application

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Create a new Python file—*quickstart-file.py*, for example. Then open it in your preferred editor or IDE.

Find the subscription key and endpoint

Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your subscription key and endpoint in the resource's **key and endpoint** page, under **resource management**.

Create variables for your Computer Vision subscription key and endpoint. Paste your subscription key and endpoint into the following code where indicated. Your Computer Vision endpoint has the form `https://<your_computer_vision_resource_name>.cognitiveservices.azure.com/`.

```
from azure.cognitiveservices.vision.computervision import ComputerVisionClient
from azure.cognitiveservices.vision.computervision.models import OperationStatusCodes
from azure.cognitiveservices.vision.computervision.models import VisualFeatureTypes
from msrest.authentication import CognitiveServicesCredentials

from array import array
import os
from PIL import Image
import sys
import time

'''
Authenticate
Authenticates your credentials and creates a client.
'''
subscription_key = "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE"
endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE"
```

> **IMPORTANT**
> Remember to remove the subscription key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. For example, Azure key vault.

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the OCR Python SDK.

NAME	DESCRIPTION
ComputerVisionClientOperationsMixin	This class directly handles all of the image operations, such as image analysis, text detection, and thumbnail generation.
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes. It implements **ComputerVisionClientOperationsMixin**.

NAME	DESCRIPTION
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of **VisualFeatureTypes** values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the OCR client library for Python:

- Authenticate the client
- Read printed and handwritten text

Authenticate the client

Instantiate a client with your endpoint and key. Create a CognitiveServicesCredentials object with your key, and use it with your endpoint to create a ComputerVisionClient object.

```
computervision_client = ComputerVisionClient(endpoint, CognitiveServicesCredentials(subscription_key))
```

I authenticated the client I ran into an issue

Read printed and handwritten text

The OCR service can read visible text in an image and convert it to a character stream. You do this in two parts.

Call the Read API

First, use the following code to call the **read** method for the given image. This returns an operation ID and starts an asynchronous process to read the content of the image.

```
'''
OCR: Read File using the Read API, extract text - remote
This example will extract text in an image, then print results, line by line.
This API call can also extract handwriting style text (not shown).
'''
print("===== Read File - remote =====")
# Get an image with text
read_image_url = "https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/master/articles/cognitive-services/Computer-vision/Images/readsample.jpg"

# Call API with URL and raw response (allows you to get the operation location)
read_response = computervision_client.read(read_image_url,  raw=True)
```

> **TIP**
> You can also read text from a local image. See the ComputerVisionClientOperationsMixin methods, such as **read_in_stream**. Or, see the sample code on GitHub for scenarios involving local images.

Get Read results

Next, get the operation ID returned from the **read** call, and use it to query the service for operation results. The following code checks the operation at one-second intervals until the results are returned. It then prints the extracted text data to the console.

```
# Get the operation location (URL with an ID at the end) from the response
read_operation_location = read_response.headers["Operation-Location"]
# Grab the ID from the URL
operation_id = read_operation_location.split("/")[-1]

# Call the "GET" API and wait for it to retrieve the results
while True:
    read_result = computervision_client.get_read_result(operation_id)
    if read_result.status not in ['notStarted', 'running']:
        break
    time.sleep(1)

# Print the detected text, line by line
if read_result.status == OperationStatusCodes.succeeded:
    for text_result in read_result.analyze_result.read_results:
        for line in text_result.lines:
            print(line.text)
            print(line.bounding_box)
print()
```

I read text I ran into an issue

Run the application

Run the application with the `python` command on your quickstart file.

```
python quickstart-file.py
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the OCR client library and use the Read API. Next, learn more about the Read API features.

Call the Read API

- OCR overview
- The source code for this sample can be found on GitHub.

Use the Optical character recognition client library to read printed and handwritten text in images.

Reference documentation | Library source code |Artifact (Maven) | Samples

Prerequisites

- An Azure subscription - Create one for free

- The current version of the Java Development Kit (JDK)
- The Gradle build tool, or another dependency manager.
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a new Gradle project

In a console window (such as cmd, PowerShell, or Bash), create a new directory for your app, and navigate to it.

```
mkdir myapp && cd myapp
```

Run the `gradle init` command from your working directory. This command will create essential build files for Gradle, including *build.gradle.kts*, which is used at runtime to create and configure your application.

```
gradle init --type basic
```

When prompted to choose a **DSL**, select **Kotlin**.

Install the client library

This quickstart uses the Gradle dependency manager. You can find the client library and information for other dependency managers on the Maven Central Repository.

Locate *build.gradle.kts* and open it with your preferred IDE or text editor. Then copy in the following build configuration. This configuration defines the project as a Java application whose entry point is the class **ComputerVisionQuickstart**. It imports the Computer Vision library.

```
plugins {
    java
    application
}
application {
    mainClass.set("ComputerVisionQuickstart")
}
repositories {
    mavenCentral()
}
dependencies {
    implementation(group = "com.microsoft.azure.cognitiveservices", name = "azure-cognitiveservices-
computervision", version = "1.0.6-beta")
}
```

Create a Java file

From your working directory, run the following command to create a project source folder:

```
mkdir -p src/main/java
```

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Navigate to the new folder and create a file called *ComputerVisionQuickstart.java*. Open it in your preferred editor or IDE.

Find the subscription key and endpoint

Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your subscription key and endpoint in the resource's **key and endpoint** page, under **resource management**.

Define the class **ComputerVisionQuickstart**. Create variables for your Computer Vision subscription key and endpoint. Paste your subscription key and endpoint into the following code where indicated. Your Computer Vision endpoint has the form `https://<your_computer_vision_resource_name>.cognitiveservices.azure.com/`.

```
import com.microsoft.azure.cognitiveservices.vision.computervision.*;
import com.microsoft.azure.cognitiveservices.vision.computervision.implementation.ComputerVisionImpl;
import com.microsoft.azure.cognitiveservices.vision.computervision.models.*;

import java.io.*;
import java.nio.file.Files;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class ComputerVisionQuickstart {

    static String subscriptionKey = "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE";
    static String endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE";
```

> **IMPORTANT**
> Remember to remove the subscription key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. For example, Azure key vault.

In the application's **main** method, add calls for the methods used in this quickstart. You'll define these later.

```
public static void main(String[] args) {

    System.out.println("\nAzure Cognitive Services Computer Vision - Java Quickstart Sample");

    // Create an authenticated Computer Vision client.
    ComputerVisionClient compVisClient = Authenticate(subscriptionKey, endpoint);

    // Read from local file
    ReadFromFile(compVisClient);
}
```

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the OCR Java SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes.

Code examples

These code snippets show you how to do the following tasks with the OCR client library for Java:

- Authenticate the client
- Read printed and handwritten text

Authenticate the client

In a new method, instantiate a ComputerVisionClient object with your endpoint and key.

```
public static ComputerVisionClient Authenticate(String subscriptionKey, String endpoint){
    return ComputerVisionManager.authenticate(subscriptionKey).withEndpoint(endpoint);
}
```

I authenticated the client I ran into an issue

Read printed and handwritten text

The OCR service can read visible text in an image and convert it to a character stream. This section defines a method, `ReadFromFile`, that takes a local file path and prints the image's text to the console.

> **TIP**
>
> You can also read text in a remote image referenced by URL. See the ComputerVision methods, such as **read**. Or, see the sample code on GitHub for scenarios involving remote images.

Set up test image

Create a **resources/** folder in the **src/main/** folder of your project, and add an image you'd like to read text from. You can download a sample image to use here.

Then add the following method definition to your **ComputerVisionQuickstart** class. Change the value of the `localFilePath` to match your image file.

```
/**
 * OCR with READ : Performs a Read Operation on a local image
 * @param client instantiated vision client
 * @param localFilePath local file path from which to perform the read operation against
 */
private static void ReadFromFile(ComputerVisionClient client) {
    System.out.println("-----------------------------------------------");

    String localFilePath = "src\\main\\resources\\myImage.png";
    System.out.println("Read with local file: " + localFilePath);
```

Call the Read API

Then, add the following code to call the **readInStreamWithServiceResponseAsync** method for the given image.

```
try {
    File rawImage = new File(localFilePath);
    byte[] localImageBytes = Files.readAllBytes(rawImage.toPath());

    // Cast Computer Vision to its implementation to expose the required methods
    ComputerVisionImpl vision = (ComputerVisionImpl) client.computerVision();

    // Read in remote image and response header
    ReadInStreamHeaders responseHeader =
            vision.readInStreamWithServiceResponseAsync(localImageBytes, null, null)
                .toBlocking()
                .single()
                .headers();
```

The following block of code extracts the operation ID from the response of the Read call. It uses this ID with a helper method to print the text read results to the console.

```
// Extract the operationLocation from the response header
String operationLocation = responseHeader.operationLocation();
System.out.println("Operation Location:" + operationLocation);

getAndPrintReadResult(vision, operationLocation);
```

Close out the try/catch block and the method definition.

```
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

Get Read results

Then, add a definition for the helper method. This method uses the operation ID from the previous step to query the read operation and get OCR results when they're available.

```
/**
 * Polls for Read result and prints results to console
 * @param vision Computer Vision instance
 * @return operationLocation returned in the POST Read response header
 */
private static void getAndPrintReadResult(ComputerVision vision, String operationLocation) throws
InterruptedException {
    System.out.println("Polling for Read results ...");

    // Extract OperationId from Operation Location
    String operationId = extractOperationIdFromOpLocation(operationLocation);

    boolean pollForResult = true;
    ReadOperationResult readResults = null;

    while (pollForResult) {
        // Poll for result every second
        Thread.sleep(1000);
        readResults = vision.getReadResult(UUID.fromString(operationId));

        // The results will no longer be null when the service has finished processing the request.
        if (readResults != null) {
            // Get request status
            OperationStatusCodes status = readResults.status();

            if (status == OperationStatusCodes.FAILED || status == OperationStatusCodes.SUCCEEDED) {
                pollForResult = false;
            }
        }
    }
```

The rest of the method parses the OCR results and prints them to the console.

```
    // Print read results, page per page
    for (ReadResult pageResult : readResults.analyzeResult().readResults()) {
        System.out.println("");
        System.out.println("Printing Read results for page " + pageResult.page());
        StringBuilder builder = new StringBuilder();

        for (Line line : pageResult.lines()) {
            builder.append(line.text());
            builder.append("\n");
        }

        System.out.println(builder.toString());
    }
}
```

Finally, add the other helper method used above, which extracts the operation ID from the initial response.

```
/**
 * Extracts the OperationId from a Operation-Location returned by the POST Read operation
 * @param operationLocation
 * @return operationId
 */
private static String extractOperationIdFromOpLocation(String operationLocation) {
    if (operationLocation != null && !operationLocation.isEmpty()) {
        String[] splits = operationLocation.split("/");

        if (splits != null && splits.length > 0) {
            return splits[splits.length - 1];
        }
    }
    throw new IllegalStateException("Something went wrong: Couldn't extract the operation id from the
operation location");
}
```

I read text I ran into an issue

Run the application

You can build the app with:

```
gradle build
```

Run the application with the `gradle run` command:

```
gradle run
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the OCR client library and use the Read API. Next, learn more about the Read API features.

Call the Read API

- OCR overview
- The source code for this sample can be found on GitHub.

Use the Optical character recognition client library to read printed and handwritten text with the Read API.

Reference documentation | Library source code | Package (npm) | Samples

Prerequisites

- An Azure subscription - Create one for free
- The current version of Node.js
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a new Node.js application

In a console window (such as cmd, PowerShell, or Bash), create a new directory for your app, and navigate to it.

```
mkdir myapp && cd myapp
```

Run the `npm init` command to create a node application with a `package.json` file.

```
npm init
```

Install the client library

Install the `ms-rest-azure` and `@azure/cognitiveservices-computervision` NPM package:

```
npm install @azure/cognitiveservices-computervision
```

Also install the async module:

```
npm install async
```

Your app's `package.json` file will be updated with the dependencies.

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Create a new file, *index.js*, and open it in a text editor.

Find the subscription key and endpoint

Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your subscription key and endpoint in the resource's **key and endpoint** page, under **resource management**.

Create variables for your Computer Vision subscription key and endpoint. Paste your subscription key and endpoint into the following code where indicated. Your Computer Vision endpoint has the form `https://<your_computer_vision_resource_name>.cognitiveservices.azure.com/`.

```
'use strict';

const async = require('async');
const fs = require('fs');
const https = require('https');
const path = require("path");
const createReadStream = require('fs').createReadStream
const sleep = require('util').promisify(setTimeout);
const ComputerVisionClient = require('@azure/cognitiveservices-computervision').ComputerVisionClient;
const ApiKeyCredentials = require('@azure/ms-rest-js').ApiKeyCredentials;

/**
 * AUTHENTICATE
 * This single client is used for all examples.
 */
const key = 'PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE';
const endpoint = 'PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE';
```

IMPORTANT

Remember to remove the subscription key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. For example, Azure key vault.

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the OCR Node.js SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to do most image operations.

Code examples

These code snippets show you how to do the following tasks with the OCR client library for Node.js:

- Authenticate the client
- Read printed and handwritten text

Authenticate the client

Instantiate a client with your endpoint and key. Create a ApiKeyCredentials object with your key and endpoint, and use it to create a ComputerVisionClient object.

```
const computerVisionClient = new ComputerVisionClient(
  new ApiKeyCredentials({ inHeader: { 'Ocp-Apim-Subscription-Key': key } }), endpoint);
```

Then, define a function `computerVision` and declare an async series with primary function and callback function. At the end of the script, you'll complete this function definition and call it.

```
function computerVision() {
  async.series([
    async function () {
```

I authenticated the client I ran into an issue

Read printed and handwritten text

The OCR service can extract the visible text in an image and convert it to a character stream. This sample uses the Read operations.

Set up test images

Save a reference of the URL of the images you want to extract text from.

```
// URL images containing printed and/or handwritten text.
// The URL can point to image files (.jpg/.png/.bmp) or multi-page files (.pdf, .tiff).
const printedTextSampleURL = 'https://moderatorsampleimages.blob.core.windows.net/samples/sample2.jpg';
const multiLingualTextURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/MultiLingual.png';
const mixedMultiPagePDFURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/MultiPageHandwrittenForm.pdf';
```

> **NOTE**
>
> You can also read text from a local image. See the ComputerVisionClient methods, such as **readInStream**. Or, see the sample code on GitHub for scenarios involving local images.

Call the Read API

Define the following fields in your function to denote the Read call status values.

```
// Status strings returned from Read API. NOTE: CASING IS SIGNIFICANT.
// Before Read 3.0, these are "Succeeded" and "Failed"
const STATUS_SUCCEEDED = "succeeded";
const STATUS_FAILED = "failed"
```

Add the code below, which calls the `readTextFromURL` function for the given images.

```
// Recognize text in printed image from a URL
console.log('Read printed text from URL...', printedTextSampleURL.split('/').pop());
const printedResult = await readTextFromURL(computerVisionClient, printedTextSampleURL);
printRecText(printedResult);

// Recognize multi-lingual text in a PNG from a URL
console.log('\nRead printed multi-lingual text in a PNG from URL...', multiLingualTextURL.split('/').pop());
const multiLingualResult = await readTextFromURL(computerVisionClient, multiLingualTextURL);
printRecText(multiLingualResult);

// Recognize printed text and handwritten text in a PDF from a URL
console.log('\nRead printed and handwritten text from a PDF from URL...',
mixedMultiPagePDFURL.split('/').pop());
const mixedPdfResult = await readTextFromURL(computerVisionClient, mixedMultiPagePDFURL);
printRecText(mixedPdfResult);
```

Define the `readTextFromURL` function. This calls the **read** method on the client object, which returns an operation ID and starts an asynchronous process to read the content of the image. Then it uses the operation ID to check the operation status until the results are returned. They it returns the extracted results.

```
// Perform read and await the result from URL
async function readTextFromURL(client, url) {
  // To recognize text in a local image, replace client.read() with readTextInStream() as shown:
  let result = await client.read(url);
  // Operation ID is last path segment of operationLocation (a URL)
  let operation = result.operationLocation.split('/').slice(-1)[0];

  // Wait for read recognition to complete
  // result.status is initially undefined, since it's the result of read
  while (result.status !== STATUS_SUCCEEDED) { await sleep(1000); result = await
client.getReadResult(operation); }
  return result.analyzeResult.readResults; // Return the first page of result. Replace [0] with the desired
page if this is a multi-page file such as .pdf or .tiff.
}
```

Then, define the helper function `printRecText`, which prints the results of the Read operations to the console.

```
// Prints all text from Read result
function printRecText(readResults) {
  console.log('Recognized text:');
  for (const page in readResults) {
    if (readResults.length > 1) {
      console.log(`==== Page: ${page}`);
    }
    const result = readResults[page];
    if (result.lines.length) {
      for (const line of result.lines) {
        console.log(line.words.map(w => w.text).join(' '));
      }
    }
    else { console.log('No recognized text.'); }
  }
}
```

I read text I ran into an issue

Close the function

Close out the `computerVision` function and call it.

```
    },
    function () {
      return new Promise((resolve) => {
        resolve();
      })
    }
  ], (err) => {
    throw (err);
  });
}

computerVision();
```

Run the application

Run the application with the `node` command on your quickstart file.

```
node index.js
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the OCR client library and use the Read API. Next, learn more about the Read API features.

Call the Read API

- OCR overview
- The source code for this sample can be found on GitHub.

Use the OCR client library to read printed and handwritten text from images.

Reference documentation | Library source code | Package

Prerequisites

- An Azure subscription - Create one for free
- The latest version of Go
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a Go project directory

In a console window (cmd, PowerShell, Terminal, Bash), create a new workspace for your Go project, named `my-app`, and navigate to it.

```
mkdir -p my-app/{src, bin, pkg}
cd my-app
```

Your workspace will contain three folders:

- **src** - This directory will contain source code and packages. Any packages installed with the `go get` command will go in this directory.
- **pkg** - This directory will contain the compiled Go package objects. These files all have an `.a` extension.
- **bin** - This directory will contain the binary executable files that are created when you run `go install`.

> **TIP**
> To learn more about the structure of a Go workspace, see the Go language documentation. This guide includes information for setting `$GOPATH` and `$GOROOT`.

Install the client library for Go

Next, install the client library for Go:

```
go get -u https://github.com/Azure/azure-sdk-for-
go/tree/master/services/cognitiveservices/v2.1/computervision
```

or if you use dep, within your repo run:

```
dep ensure -add https://github.com/Azure/azure-sdk-for-
go/tree/master/services/cognitiveservices/v2.1/computervision
```

Create a Go application

Next, create a file in the **src** directory named `sample-app.go`:

```
cd src
touch sample-app.go
```

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Open `sample-app.go` in your preferred IDE or text editor.

Declare a context at the root of your script. You'll need this object to execute most Computer Vision function calls.

Find the subscription key and endpoint

Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your subscription key and endpoint in the resource's **key and endpoint** page, under **resource management**.

Create variables for your Computer Vision subscription key and endpoint. Paste your subscription key and endpoint into the following code where indicated. Your Computer Vision endpoint has the form `https://<your_computer_vision_resource_name>.cognitiveservices.azure.com/`.

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/Azure/azure-sdk-for-go/services/cognitiveservices/v2.0/computervision"
    "github.com/Azure/go-autorest/autorest"
    "io"
    "log"
    "os"
    "strings"
    "time"
)

// Declare global so don't have to pass it to all of the tasks.
var computerVisionContext context.Context

func main() {
    computerVisionKey := "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE"
    endpointURL := "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE"
```

IMPORTANT

Remember to remove the subscription key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. For example, Azure key vault.

Next, you'll begin adding code to carry out different OCR operations.

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the OCR Go SDK.

NAME	DESCRIPTION
BaseClient	This class is needed for all Computer Vision functionality, such as image analysis and text reading. You instantiate it with your subscription information, and you use it to do most image operations.
ReadOperationResult	This type contains the results of a Batch Read operation.

Code examples

These code snippets show you how to do the following tasks with the OCR client library for Go:

- Authenticate the client
- Read printed and handwritten text

Authenticate the client

> **NOTE**
>
> This step assumes you've created environment variables for your Computer Vision key and endpoint, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT` respectively.

Create a `main` function and add the following code to it to instantiate a client with your endpoint and key.

```
/*
 * Configure the Computer Vision client
 */
computerVisionClient := computervision.New(endpointURL);
computerVisionClient.Authorizer = autorest.NewCognitiveServicesAuthorizer(computerVisionKey)

computerVisionContext = context.Background()
/*
 * END - Configure the Computer Vision client
 */
```

I authenticated the client I ran into an issue

Read printed and handwritten text

The OCR service can read visible text in an image and convert it to a character stream. The code in this section defines a function, `RecognizeTextReadAPIRemoteImage`, which uses the client object to detect and extract printed or handwritten text in the image.

Add the sample image reference and function call in your `main` function.

```
// Analyze text in an image, remote
BatchReadFileRemoteImage(computerVisionClient, printedImageURL)
```

> **TIP**
>
> You can also extract text from a local image. See the BaseClient methods, such as **BatchReadFileInStream**. Or, see the sample code on GitHub for scenarios involving local images.

Call the Read API

Define the new function for reading text, `RecognizeTextReadAPIRemoteImage`. Add the code below, which calls the **BatchReadFile** method for the given image. This method returns an operation ID and starts an asynchronous process to read the content of the image.

```
func BatchReadFileRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("BATCH READ FILE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // The response contains a field called "Operation-Location",
    // which is a URL with an ID that you'll use for GetReadOperationResult to access OCR results.
    textHeaders, err := client.BatchReadFile(computerVisionContext, remoteImage)
    if err != nil { log.Fatal(err) }

    // Use ExtractHeader from the autorest library to get the Operation-Location URL
    operationLocation := autorest.ExtractHeaderValue("Operation-Location", textHeaders.Response)

    numberOfCharsInOperationId := 36
    operationId := string(operationLocation[len(operationLocation)-numberOfCharsInOperationId :
len(operationLocation)])
```

Get Read results

Next, get the operation ID returned from the **BatchReadFile** call, and use it with the **GetReadOperationResult** method to query the service for operation results. The following code checks the operation at one-second intervals until the results are returned. It then prints the extracted text data to the console.

```
readOperationResult, err := client.GetReadOperationResult(computerVisionContext, operationId)
if err != nil { log.Fatal(err) }

// Wait for the operation to complete.
i := 0
maxRetries := 10

fmt.Println("Recognizing text in a remote image with the batch Read API ...")
for readOperationResult.Status != computervision.Failed &&
        readOperationResult.Status != computervision.Succeeded {
    if i >= maxRetries {
        break
    }
    i++

    fmt.Printf("Server status: %v, waiting %v seconds...\n", readOperationResult.Status, i)
    time.Sleep(1 * time.Second)

    readOperationResult, err = client.GetReadOperationResult(computerVisionContext, operationId)
    if err != nil { log.Fatal(err) }
}
```

Display Read results

Add the following code to parse and display the retrieved text data, and finish the function definition.

```
// Display the results.
fmt.Println()
for _, recResult := range *(readOperationResult.RecognitionResults) {
    for _, line := range *recResult.Lines {
        fmt.Println(*line.Text)
    }
}
```

I read text I ran into an issue

Run the application

Run the application from your application directory with the `go run` command.

```
go run sample-app.go
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the OCR client library and use the Read API. Next, learn more about the Read API features.

Call the Read API

- OCR overview
- The source code for this sample can be found on GitHub.

Use the Optical character recognition REST API to read printed and handwritten text.

> **NOTE**
> This quickstart uses cURL commands to call the REST API. You can also call the REST API using a programming language. See the GitHub samples for examples in C#, Python, Java, JavaScript, and Go.

Prerequisites

- An Azure subscription - Create one for free
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- cURL installed

Read printed and handwritten text

The OCR service can read visible text in an image and convert it to a character stream. For more information on text recognition, see the Optical character recognition (OCR) overview.

Call the Read API

To create and run the sample, do the following steps:

1. Copy the following command into a text editor.
2. Make the following changes in the command where needed:

a. Replace the value of `<subscriptionKey>` with your subscription key.
b. Replace the first part of the request URL (`westcentralus`) with the text in your own endpoint URL.

> **NOTE**
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

c. Optionally, change the image URL in the request body (`https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png\`) to the URL of a different image to be analyzed.

3. Open a command prompt window.
4. Paste the command from the text editor into the command prompt window, and then run the command.

```
curl -v -X POST "https://westcentralus.api.cognitive.microsoft.com/vision/v3.2/read/analyze" -H "Content-Type: application/json" -H "Ocp-Apim-Subscription-Key: <subscription key>" --data-ascii "
{\"url\":\"https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Atomist_quote_from_Democritus.png/338px-Atomist_quote_from_Democritus.png\"}"
```

The response will include an `Operation-Location` header, whose value is a unique URL. You use this URL to query the results of the Read operation. The URL expires in 48 hours.

Get Read results

1. Copy the following command into your text editor.
2. Replace the URL with the `Operation-Location` value you copied in the previous step.
3. Make the following changes in the command where needed:
   a. Replace the value of `<subscriptionKey>` with your subscription key.
4. Open a command prompt window.
5. Paste the command from the text editor into the command prompt window, and then run the command.

```
curl -v -X GET "https://westcentralus.api.cognitive.microsoft.com/vision/v3.2/read/analyzeResults/{operationId}" -H "Ocp-Apim-Subscription-Key: {subscription key}" --data-ascii "{body}"
```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the command prompt window, similar to the following example:

```
{
  "status": "succeeded",
  "createdDateTime": "2021-04-08T21:56:17.6819115+00:00",
  "lastUpdatedDateTime": "2021-04-08T21:56:18.4161316+00:00",
  "analyzeResult": {
    "version": "3.2",
    "readResults": [
      {
        "page": 1,
        "angle": 0,
        "width": 338,
        "height": 479,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              25,
              14,
              318,
              14,
              318,
              59,
              25,
              59
            ],
            "text": "NOTHING",
            "appearance": {
              "style": {
                "name": "other",
                "confidence": 0.971
              }
            },
            "words": [
              {
                "boundingBox": [
                  27,
                  15,
                  294,
                  15,
                  294,
                  60,
                  27,
                  60
                ],
                "text": "NOTHING",
                "confidence": 0.994
              }
            ]
          }
        ]
      }
    ]
  }
}
```


Next steps

In this quickstart, you learned how to call the Read REST API. Next, learn more about the Read API features.

Call the Read API

- OCR overview

Call the Read API

6/16/2021 • 3 minutes to read • Edit Online

In this guide, you'll learn how to call the Read API to extract text from images. You'll learn the different ways you can configure the behavior of this API to meet your needs.

This guide assumes you have already create a Computer Vision resource and obtained a subscription key and endpoint URL. If you haven't, follow a quickstart to get started.

Submit data to the service

You submit either a local image or a remote image to the Read API. For local, you put the binary image data in the HTTP request body. For remote, you specify the image's URL by formatting the request body like the following: `{"url":"http://example.com/images/test.jpg"}`.

The Read API's Read call takes an image or PDF document as the input and extracts text asynchronously.

```
https://{endpoint}/vision/v3.2/read/analyze[?language][&pages][&readingOrder]
```

The call returns with a response header field called `Operation-Location`. The `Operation-Location` value is a URL that contains the Operation ID to be used in the next step.

RESPONSE HEADER	EXAMPLE VALUE
Operation-Location	`https://cognitiveservice/vision/v3.2/read/analyzeResults/49a3632-fc4b-4387-aa06-090cfbf0064f`

NOTE

Billing

The Computer Vision pricing page includes the pricing tier for Read. Each analyzed image or page is one transaction. If you call the operation with a PDF or TIFF document containing 100 pages, the Read operation will count it as 100 transactions and you will be billed for 100 transactions. If you made 50 calls to the operation and each call submitted a document with 100 pages, you will be billed for 50 X 100 = 5000 transactions.

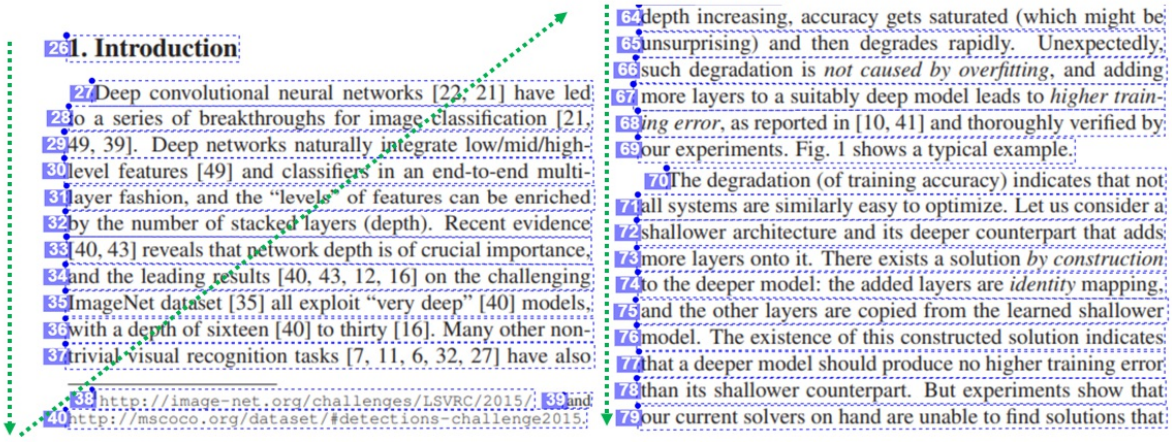
Determine how to process the data

Language specification

The Read call has an optional request parameter for language. Read supports auto language identification and multilingual documents, so only provide a language code if you would like to force the document to be processed as that specific language.

Natural reading order output (Latin languages only)

Specify the order in which the text lines are output with the `readingOrder` query parameter. Use `natural` for a more human-friendly reading order output as shown in the following example. This feature is only supported for Latin languages.



Select page(s) or page ranges for text extraction

For large multi-page documents, use the `pages` query parameter to specify page numbers or page ranges to extract text from only those pages. The following example shows a document with 10 pages, with text extracted for both cases - all pages (1-10) and selected pages (3-6).

All pages (1-10, default)

Selected pages (3-6)

Get results from the service

The second step is to call Get Read Results operation. This operation takes as input the operation ID that was created by the Read operation.

```
https://{endpoint}/vision/v3.2/read/analyzeResults/{operationId}
```

It returns a JSON response that contains a **status** field with the following possible values.

VALUE	MEANING
`notStarted`	The operation has not started.
`running`	The operation is being processed.

VALUE	MEANING
`failed`	The operation has failed.
`succeeded`	The operation has succeeded.

You call this operation iteratively until it returns with the **succeeded** value. Use an interval of 1 to 2 seconds to avoid exceeding the requests per second (RPS) rate.

> **NOTE**
> The free tier limits the request rate to 20 calls per minute. The paid tier allows 10 requests per second (RPS) that can be increased upon request. Note your Azure resource identfier and region, and open an Azure support ticket or contact your account team to request a higher request per second (RPS) rate.

When the **status** field has the `succeeded` value, the JSON response contains the extracted text content from your image or document. The JSON response maintains the original line groupings of recognized words. It includes the extracted text lines and their bounding box coordinates. Each text line includes all extracted words with their coordinates and confidence scores.

> **NOTE**
> The data submitted to the `Read` operation are temporarily encrypted and stored at rest for a short duration, and then deleted. This lets your applications retrieve the extracted text as part of the service response.

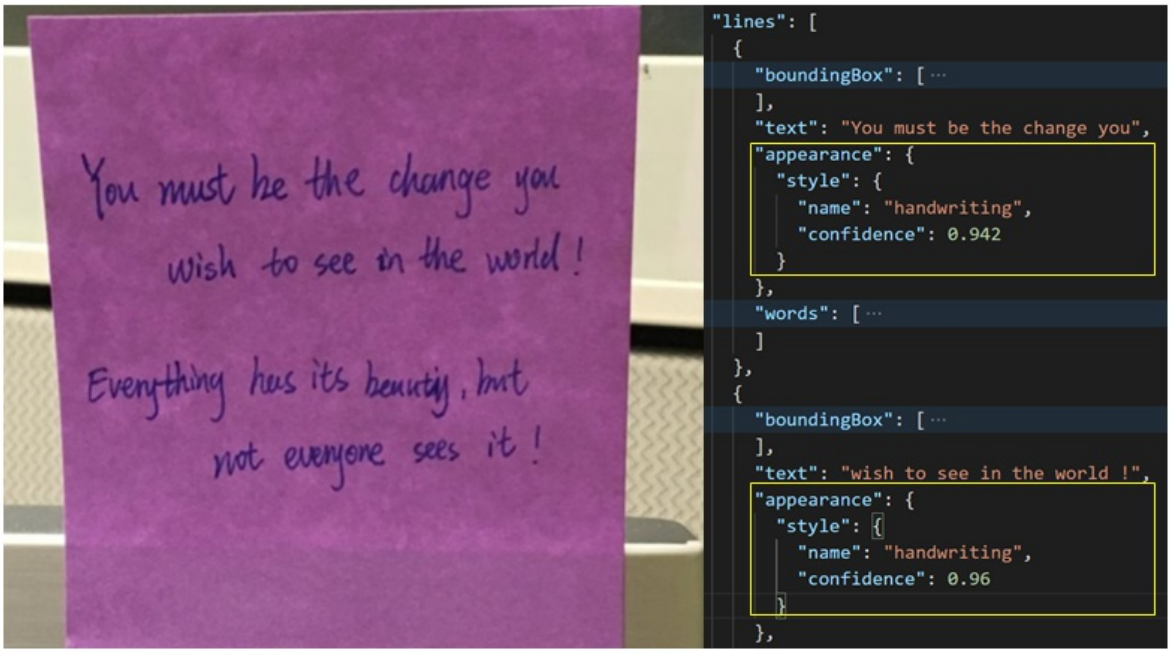
Sample JSON output

See the following example of a successful JSON response:

```
{
  "status": "succeeded",
  "createdDateTime": "2021-02-04T06:32:08.2752706+00:00",
  "lastUpdatedDateTime": "2021-02-04T06:32:08.7706172+00:00",
  "analyzeResult": {
    "version": "3.2",
    "readResults": [
      {
        "page": 1,
        "angle": 2.1243,
        "width": 502,
        "height": 252,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              58,
              42,
              314,
              59,
              311,
              123,
              56,
              121
            ],
            "text": "Tabs vs",
            "appearance": {
              "style": {
                "name": "handwriting",
                "confidence": 0.96
              }
            },
            "words": [
              {
                "boundingBox": [
                  68,
                  44,
                  225,
                  59,
                  224,
                  122,
                  66,
                  123
                ],
                "text": "Tabs",
                "confidence": 0.933
              },
              {
                "boundingBox": [
                  241,
                  61,
                  314,
                  72,
                  314,
                  123,
                  239,
                  122
                ],
                "text": "vs",
                "confidence": 0.977
              }
            ]
          }
        ]
      }
    ]
  }
}
```

Handwritten classification for text lines (Latin languages only)

The response includes classifying whether each text line is of handwriting style or not, along with a confidence score. This feature is only supported for Latin languages. The following example shows the handwritten classification for the text in the image.



Next steps

To try out the REST API, go to the Read API Reference.

Upgrade from Read v2.x to Read v3.x

4/27/2021 • 5 minutes to read • Edit Online

This guide shows how to upgrade your existing container or cloud API code from Read v2.x to Read v3.x.

Determine your API path

Use the following table to determine the **version string** in the API path based on the Read 3.x version you are migrating to.

PRODUCT TYPE	VERSION	VERSION STRING IN 3.X API PATH
Service	Read 3.0, 3.1, or 3.2	**v3.0**, **v3.1**, or **v3.2** respectively
Service	Read 3.2 preview	**v3.2-preview.1**
Container	Read 3.0 preview or Read 3.1 preview	**v3.0** or **v3.1-preview.2** respectively

Next, use the following sections to narrow your operations and replace the **version string** in your API path with the value from the table. For example, for **Read v3.2 preview** cloud and container versions, update the API path to **https://{endpoint}/vision/v3.2-preview.1/read/analyze[?language]**.

Service/Container

`Batch Read File`

READ 2.X	READ 3.X
https://{endpoint}/vision/**v2.0/read/core/asyncBatchAnalyze**	https://{endpoint}/vision/<**version string**>/read/analyze[?language]

A new optional *language* parameter is available. If you do not know the language of your document, or it may be multilingual, don't include it.

`Get Read Results`

READ 2.X	READ 3.X
https://{endpoint}/vision/**v2.0/read/operations**/{operationId}	https://{endpoint}/vision/<**version string**>/read/analyzeResults/{operationId}

`Get Read Operation Result` **status flag**

When the call to `Get Read Operation Result` is successful, it returns a status string field in the JSON body.

READ 2.X	READ 3.X
`"NotStarted"`	`"notStarted"`
`"Running"`	`"running"`

READ 2.X	READ 3.X
`"Failed"`	`"failed"`
`"Succeeded"`	`"succeeded"`

API response (JSON)

Note the following changes to the json:

- In v2.x, `Get Read Operation Result` will return the OCR recognition json when the status is `Succeeded"`. In v3.0, this field is `succeeded`.
- To get the root for page array, change the json hierarchy from `recognitionResults` to `analyzeResult`/`readResults`. The per-page line and words json hierarchy remains unchanged, so no code changes are required.
- The page angle `clockwiseOrientation` has been renamed to `angle` and the range has been changed from 0 - 360 degrees to -180 to 180 degrees. Depending on your code, you may or may not have to makes changes as most math functions can handle either range.

The v3.0 API also introduces the following improvements you can optionally leverage:

- `createdDateTime` and `lastUpdatedDateTime` are added so you can track the duration of processing. See documentation for more details.
- `version` tells you the version of the API used to generate results
- A per-word `confidence` has been added. This value is calibrated so that a value 0.95 means that there is a 95% chance the recognition is correct. The confidence score can be used to select which text to send to human review.

In 2.X, the output format is as follows:

```
    {
        {
                "status": "Succeeded",
                "recognitionResults": [
                    {
                    "page": 1,
                    "language": "en",
                    "clockwiseOrientation": 349.59,
                    "width": 2661,
                    "height": 1901,
                    "unit": "pixel",
                    "lines": [
                        {
                        "boundingBox": [
                            67,
                            646,
                            2582,
                            713,
                            2580,
                            876,
                            67,
                            821
                        ],
                        "text": "The quick brown fox jumps",
                        "words": [
                            {
                            "boundingBox": [
                                143,
                                650,
                                435,
                                661,
                                436,
                                823,
                                144,
                                824
                            ],
                            "text": "The",
                            },
            // The rest of result is omitted for brevity

}
```

In v3.0, it has been adjusted:

```
{
    {
        "status": "succeeded",
        "createdDateTime": "2020-05-28T05:13:21Z",
        "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
        "analyzeResult": {
        "version": "3.0.0",
        "readResults": [
            {
            "page": 1,
            "language": "en",
            "angle": 0.8551,
            "width": 2661,
            "height": 1901,
            "unit": "pixel",
            "lines": [
                {
                "boundingBox": [
                    67,
                    646,
                    2582,
                    713,
                    2580,
                    876,
                    67,
                    821
                ],
                "text": "The quick brown fox jumps",
                "words": [
                    {
                    "boundingBox": [
                        143,
                        650,
                        435,
                        661,
                        436,
                        823,
                        144,
                        824
                    ],
                    "text": "The",
                    "confidence": 0.958
                    },
    // The rest of result is omitted for brevity

}
```


Service only

`Recognize Text`

`Recognize Text` is a *preview* operation which is being *deprecated in all versions of Computer Vision API*. You must migrate from `Recognize Text` to `Read` (v3.0) or `Batch Read File` (v2.0, v2.1). v3.0 of `Read` includes newer, better models for text recognition and additional features, so it is recommended. To upgrade from `Recognize Text` to `Read`:

RECOGNIZE TEXT 2.X	READ 3.X
https://{endpoint}/vision/**v2.0/recognizeText[?mode]**	https://{endpoint}/vision/<**version string**>/read/analyze[?language]

The *mode* parameter is not supported in `Read`. Both handwritten and printed text will automatically be

supported.

A new optional *language* parameter is available in v3.0. If you do not know the language of your document, or it may be multilingual, don't include it.

`Get Recognize Text Operation Result`

RECOGNIZE TEXT 2.X	READ 3.X
https://{endpoint}/vision/**v2.0/textOperations/**{operationId}	https://{endpoint}/vision/<**version string**>/read/analyzeResults/{operationId}

`Get Recognize Text Operation Result` **status flags**

When the call to `Get Recognize Text Operation Result` is successful, it returns a status string field in the JSON body.

RECOGNIZE TEXT 2.X	READ 3.X
`"NotStarted"`	`"notStarted"`
`"Running"`	`"running"`
`"Failed"`	`"failed"`
`"Succeeded"`	`"succeeded"`

API response (JSON)

Note the following changes to the json:

- In v2.x, `Get Read Operation Result` will return the OCR recognition json when the status is `Succeeded`. In v3.x, this field is `succeeded`.
- To get the root for page array, change the json hierarchy from `recognitionResult` to `analyzeResult`/`readResults`. The per-page line and words json hierarchy remains unchanged, so no code changes are required.

The v3.0 API also introduces the following improvements you can optionally leverage. See the API reference for more details:

- `createdDateTime` and `lastUpdatedDateTime` are added so you can track the duration of processing. See documentation for more details.
- `version` tells you the version of the API used to generate results
- A per-word `confidence` has been added. This value is calibrated so that a value 0.95 means that there is a 95% chance the recognition is correct. The confidence score can be used to select which text to send to human review.
- `angle` general orientation of the text in clockwise direction, measured in degrees between (-180, 180].
- `width` and `"height"` give you the dimensions of your document, and `"unit"` provides the unit of those dimensions (pixels or inches, depending on document type.)
- `page` multipage documents are supported
- `language` the input language of the document (from the optional *language* parameter.)

In 2.X, the output format is as follows:

```
{
    {
            "status": "Succeeded",
            "recognitionResult": [
                {
                "lines": [
                    {
                    "boundingBox": [
                        67,
                        646,
                        2582,
                        713,
                        2580,
                        876,
                        67,
                        821
                    ],
                    "text": "The quick brown fox jumps",
                    "words": [
                        {
                        "boundingBox": [
                            143,
                            650,
                            435,
                            661,
                            436,
                            823,
                            144,
                            824
                        ],
                        "text": "The",
                        },
        // The rest of result is omitted for brevity

}
```

In v3.x, it has been adjusted:

```
{
    {
        "status": "succeeded",
        "createdDateTime": "2020-05-28T05:13:21Z",
        "lastUpdatedDateTime": "2020-05-28T05:13:22Z",
        "analyzeResult": {
        "version": "3.0.0",
        "readResults": [
            {
            "page": 1,
            "angle": 0.8551,
            "width": 2661,
            "height": 1901,
            "unit": "pixel",
            "lines": [
                {
                "boundingBox": [
                    67,
                    646,
                    2582,
                    713,
                    2580,
                    876,
                    67,
                    821
                ],
                "text": "The quick brown fox jumps",
                "words": [
                    {
                    "boundingBox": [
                        143,
                        650,
                        435,
                        661,
                        436,
                        823,
                        144,
                        824
                    ],
                    "text": "The",
                    "confidence": 0.958
                    },
    // The rest of result is omitted for brevity

}
```


Container only

`Synchronous Read`

READ 2.0	READ 3.X
https://{endpoint}/vision/**v2.0/read/core/Analyze**	https://{endpoint}/vision/<**version string**>/read/syncAnalyze[?language]

Install Read OCR Docker containers

5/3/2021 • 15 minutes to read • Edit Online

> **NOTE**
> Starting September 22nd 2020, most gated containers are hosted on the Microsoft Container Registry, and downloading them doesn't require you to use the docker login command. You will still need to complete an online request to run the container. See the **Request approval to run the container** section later in the article for more information.

Containers enable you to run the Computer Vision APIs in your own environment. Containers are great for specific security and data governance requirements. In this article you'll learn how to download, install, and run Computer Vision containers.

The *Read* OCR container allows you to extract printed and handwritten text from images and documents with support for JPEG, PNG, BMP, PDF, and TIFF file formats. For more information, see the Read API how-to guide.

Read 3.2 container

The Read 3.2 OCR container provides:

- New models for enhanced accuracy.
- Support for multiple languages within the same document.
- Support for a total of 73 languages. See the full list of OCR-supported languages.
- A single operation for both documents and images.
- Support for larger documents and images.
- Confidence scores.
- Support for documents with both print and handwritten text.
- Ability to extract text from only selected page(s) in a document.
- Choose text line output order from default to a more natural reading order for Latin languages only.
- Text line classification as handwritten style or not for Latin languages only.

If you're using Read 2.0 containers today, see the migration guide to learn about changes in the new versions.

Prerequisites

You must meet the following prerequisites before using the containers:

REQUIRED	PURPOSE
Docker Engine	You need the Docker Engine installed on a host computer. Docker provides packages that configure the Docker environment on macOS, Windows, and Linux. For a primer on Docker and container basics, see the Docker overview. Docker must be configured to allow the containers to connect with and send billing data to Azure. **On Windows**, Docker must also be configured to support Linux containers.

REQUIRED	PURPOSE
Familiarity with Docker	You should have a basic understanding of Docker concepts, like registries, repositories, containers, and container images, as well as knowledge of basic `docker` commands.
Computer Vision resource	In order to use the container, you must have: An Azure **Computer Vision** resource and the associated API key the endpoint URI. Both values are available on the Overview and Keys pages for the resource and are required to start the container. **{API_KEY}**: One of the two available resource keys on the **Keys** page **{ENDPOINT_URI}**: The endpoint as provided on the **Overview** page

If you don't have an Azure subscription, create a free account before you begin.

Request approval to run the container

Fill out and submit the request form to request approval to run the container.

The form requests information about you, your company, and the user scenario for which you'll use the container. After you submit the form, the Azure Cognitive Services team will review it and email you with a decision.

> **IMPORTANT**
>
> - On the form, you must use an email address associated with an Azure subscription ID.
> - The Azure resource you use to run the container must have been created with the approved Azure subscription ID.
> - Check your email (both inbox and junk folders) for updates on the status of your application from Microsoft.

After you're approved, you will be able to run the container after downloading it from the Microsoft Container Registry (MCR), described later in the article.

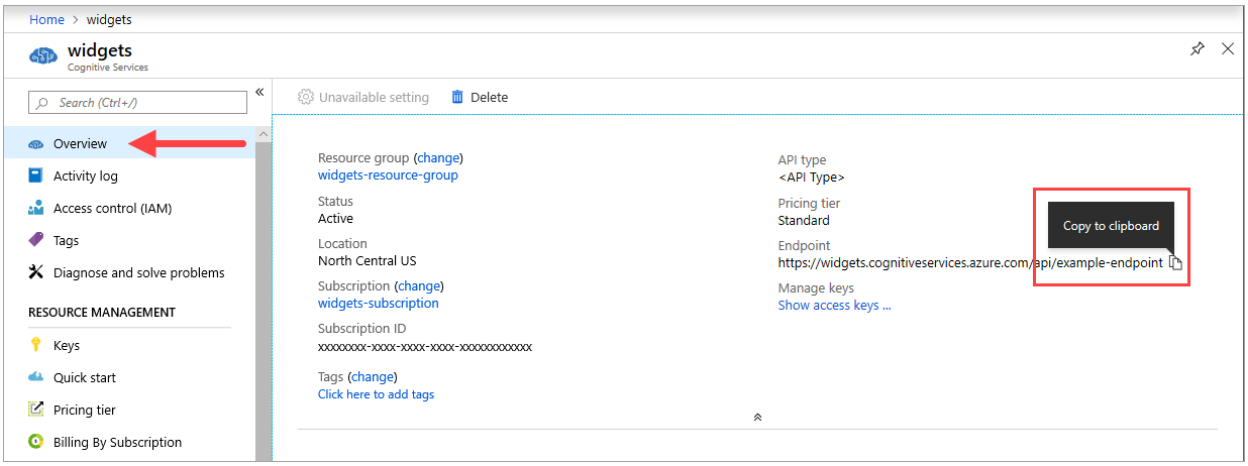
You won't be able to run the container if your Azure subscription has not been approved.

Gathering required parameters

There are three primary parameters for all Cognitive Services' containers that are required. The end-user license agreement (EULA) must be present with a value of `accept`. Additionally, both an Endpoint URL and API Key are needed.

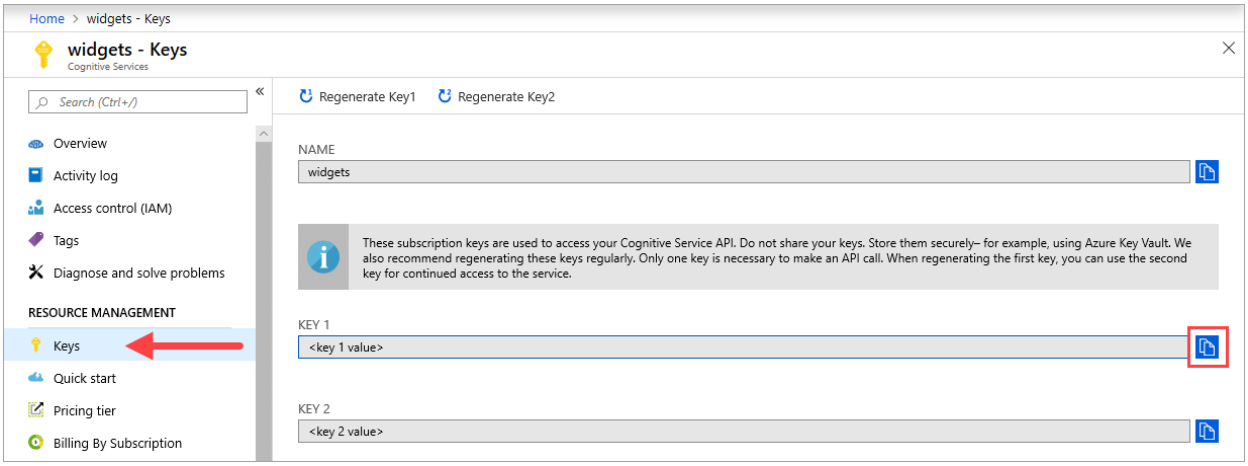
Endpoint URI `{ENDPOINT_URI}`

The **Endpoint** URI value is available on the Azure portal *Overview* page of the corresponding Cognitive Service resource. Navigate to the *Overview* page, hover over the Endpoint, and a `Copy to clipboard` icon will appear. Copy and use where needed.



Keys `{API_KEY}`

This key is used to start the container, and is available on the Azure portal's Keys page of the corresponding Cognitive Service resource. Navigate to the *Keys* page, and click on the `Copy to clipboard` icon.



> **IMPORTANT**
>
> These subscription keys are used to access your Cognitive Service API. Do not share your keys. Store them securely, for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

The host computer

The host is a x64-based computer that runs the Docker container. It can be a computer on your premises or a Docker hosting service in Azure, such as:

- Azure Kubernetes Service.
- Azure Container Instances.
- A Kubernetes cluster deployed to Azure Stack. For more information, see Deploy Kubernetes to Azure Stack.

Advanced Vector Extension support

The **host** computer is the computer that runs the docker container. The host *must support* Advanced Vector Extensions (AVX2). You can check for AVX2 support on Linux hosts with the following command:

```
grep -q avx2 /proc/cpuinfo && echo AVX2 supported || echo No AVX2 support detected
```

> **WARNING**
> The host computer is *required* to support AVX2. The container *will not* function correctly without AVX2 support.

Container requirements and recommendations

> **NOTE**
> The requirements and recommendations are based on benchmarks with a single request per second, using an 8-MB image of a scanned business letter that contains 29 lines and a total of 803 characters.

The following table describes the minimum and recommended allocation of resources for each Read OCR container.

CONTAINER	MINIMUM	RECOMMENDED
Read 2.0-preview	1 core, 8-GB memory	8 cores, 16-GB memory
Read 3.2	8 cores, 16-GB memory	8 cores, 24-GB memory

- Each core must be at least 2.6 gigahertz (GHz) or faster.

Core and memory correspond to the `--cpus` and `--memory` settings, which are used as part of the `docker run` command.

Get the container image with `docker pull`

Container images for Read are available.

CONTAINER	CONTAINER REGISTRY / REPOSITORY / IMAGE NAME
Read 2.0-preview	`mcr.microsoft.com/azure-cognitive-services/vision/read:2.0-preview`
Read 3.2	`mcr.microsoft.com/azure-cognitive-services/vision/read:3.2`

Use the `docker pull` command to download a container image.

Docker pull for the Read OCR container

- Version 3.2
- Version 2.0-preview

```
docker pull mcr.microsoft.com/azure-cognitive-services/vision/read:3.2
```

TIP

You can use the docker images command to list your downloaded container images. For example, the following command lists the ID, repository, and tag of each downloaded container image, formatted as a table:

```
docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"

IMAGE ID         REPOSITORY                TAG
<image-id>       <repository-path/name>    <tag-name>
```

How to use the container

Once the container is on the host computer, use the following process to work with the container.

1. Run the container, with the required billing settings. More examples of the `docker run` command are available.
2. Query the container's prediction endpoint.

Run the container with `docker run`

Use the docker run command to run the container. Refer to gathering required parameters for details on how to get the `{ENDPOINT_URI}` and `{API_KEY}` values.

Examples of the `docker run` command are available.

- Version 3.2
- Version 2.0-preview

```
docker run --rm -it -p 5000:5000 --memory 18g --cpus 8 \
mcr.microsoft.com/azure-cognitive-services/vision/read:3.2 \
Eula=accept \
Billing={ENDPOINT_URI} \
ApiKey={API_KEY}
```

This command:

- Runs the Read OCR container from the container image.
- Allocates 8 CPU core and 18 gigabytes (GB) of memory.
- Exposes TCP port 5000 and allocates a pseudo-TTY for the container.
- Automatically removes the container after it exits. The container image is still available on the host computer.

You can alternatively run the container using environment variables:

```
docker run --rm -it -p 5000:5000 --memory 18g --cpus 8 \
--env Eula=accept \
--env Billing={ENDPOINT_URI} \
--env ApiKey={API_KEY} \
mcr.microsoft.com/azure-cognitive-services/vision/read:3.2
```

More examples of the `docker run` command are available.

IMPORTANT

The `Eula`, `Billing`, and `ApiKey` options must be specified to run the container; otherwise, the container won't start. For more information, see Billing.

If you need higher throughput (for example, when processing multi-page files), consider deploying multiple containers on a Kubernetes cluster, using Azure Storage and Azure Queue.

If you're using Azure Storage to store images for processing, you can create a connection string to use when calling the container.

To find your connection string:

1. Navigate to **Storage accounts** on the Azure portal, and find your account.
2. Click on **Access keys** in the left navigation list.
3. Your connection string will be located below **Connection string**

Run multiple containers on the same host

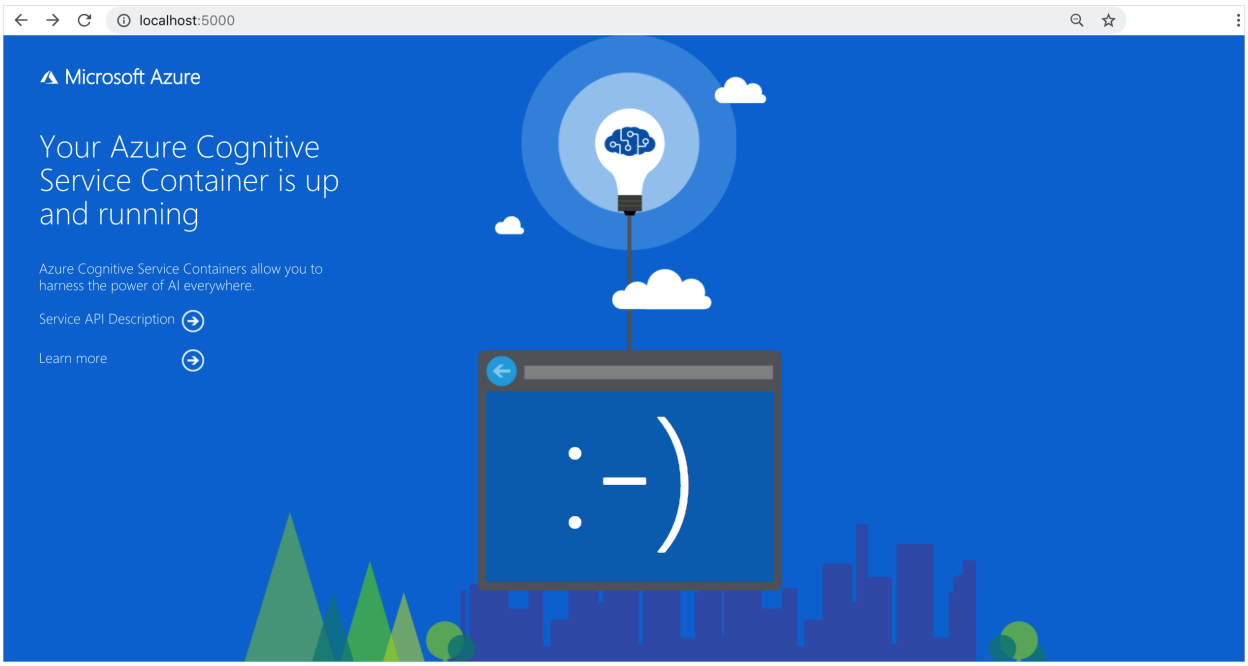
If you intend to run multiple containers with exposed ports, make sure to run each container with a different exposed port. For example, run the first container on port 5000 and the second container on port 5001.

You can have this container and a different Azure Cognitive Services container running on the HOST together. You also can have multiple containers of the same Cognitive Services container running.

Validate that a container is running

There are several ways to validate that the container is running. Locate the *External IP* address and exposed port of the container in question, and open your favorite web browser. Use the various request URLs below to validate the container is running. The example request URLs listed below are `http://localhost:5000`, but your specific container may vary. Keep in mind that you're to rely on your container's *External IP* address and exposed port.

REQUEST URL	PURPOSE
`http://localhost:5000/`	The container provides a home page.
`http://localhost:5000/ready`	Requested with GET, this provides a verification that the container is ready to accept a query against the model. This request can be used for Kubernetes liveness and readiness probes.
`http://localhost:5000/status`	Also requested with GET, this verifies if the api-key used to start the container is valid without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes.
`http://localhost:5000/swagger`	The container provides a full set of documentation for the endpoints and a **Try it out** feature. With this feature, you can enter your settings into a web-based HTML form and make the query without having to write any code. After the query returns, an example CURL command is provided to demonstrate the HTTP headers and body format that's required.



Query the container's prediction endpoint

The container provides REST-based query prediction endpoint APIs.

- Version 3.2
- Version 2.0-preview

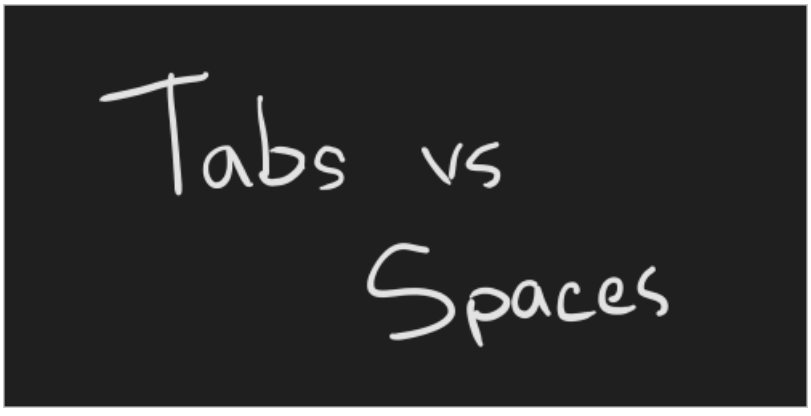
Use the host, `http://localhost:5000`, for container APIs. You can view the Swagger path at: `http://localhost:5000/swagger/vision-v3.2-read/swagger.json`.

Asynchronous read

- Version 3.2
- Version 2.0-preview

You can use the `POST /vision/v3.2/read/analyze` and `GET /vision/v3.2/read/operations/{operationId}` operations in concert to asynchronously read an image, similar to how the Computer Vision service uses those corresponding REST operations. The asynchronous POST method will return an `operationId` that is used as the identifer to the HTTP GET request.

From the swagger UI, select the `Analyze` to expand it in the browser. Then select **Try it out** > **Choose file**. In this example, we'll use the following image:



When the asynchronous POST has run successfully, it returns an **HTTP 202** status code. As part of the response, there is an `operation-location` header that holds the result endpoint for the request.

```
content-length: 0
date: Fri, 04 Sep 2020 16:23:01 GMT
operation-location: http://localhost:5000/vision/v3.2/read/operations/a527d445-8a74-4482-8cb3-c98a65ec7ef9
server: Kestrel
```

The `operation-location` is the fully qualified URL and is accessed via an HTTP GET. Here is the JSON response from executing the `operation-location` URL from the preceding image:


```
{
  "status": "succeeded",
  "createdDateTime": "2021-02-04T06:32:08.2752706+00:00",
  "lastUpdatedDateTime": "2021-02-04T06:32:08.7706172+00:00",
  "analyzeResult": {
    "version": "3.2.0",
    "readResults": [
      {
        "page": 1,
        "angle": 2.1243,
        "width": 502,
        "height": 252,
        "unit": "pixel",
        "lines": [
          {
            "boundingBox": [
              58,
              42,
              314,
              59,
              311,
              123,
              56,
              121
            ],
            "text": "Tabs vs",
            "appearance": {
              "style": {
                "name": "handwriting",
                "confidence": 0.96
              }
            },
            "words": [
              {
                "boundingBox": [
                  68,
                  44,
                  225,
                  59,
                  224,
                  122,
                  66,
                  123
                ],
                "text": "Tabs",
                "confidence": 0.933
              },
              {
                "boundingBox": [
                  241,
                  61,
                  314,
                  72,
                  314,
                  123,
                  239,
                  122
                ]
```

```
                ],
                "text": "vs",
                "confidence": 0.977
              }
            ]
          },
          {
            "boundingBox": [
              286,
              171,
              415,
              165,
              417,
              197,
              287,
              201
            ],
            "text": "paces",
            "appearance": {
              "style": {
                "name": "handwriting",
                "confidence": 0.746
              }
            },
            "words": [
              {
                "boundingBox": [
                  286,
                  179,
                  404,
                  166,
                  405,
                  198,
                  290,
                  201
                ],
                "text": "paces",
                "confidence": 0.938
              }
            ]
          }
        ]
      }
    ]
  }
}
```


IMPORTANT

If you deploy multiple Read OCR containers behind a load balancer, for example, under Docker Compose or Kubernetes, you must have an external cache. Because the processing container and the GET request container might not be the same, an external cache stores the results and shares them across containers. For details about cache settings, see Configure Computer Vision Docker containers.

Synchronous read

You can use the following operation to synchronously read an image.

- Version 3.2
- Version 2.0-preview

`POST /vision/v3.2/read/syncAnalyze`

When the image is read in its entirety, then and only then does the API return a JSON response. The only

exception to this is if an error occurs. When an error occurs the following JSON is returned:

```
{
    "status": "Failed"
}
```

The JSON response object has the same object graph as the asynchronous version. If you're a JavaScript user and want type safety, consider using TypeScript to cast the JSON response.

For an example use-case, see the TypeScript sandbox here and select **Run** to visualize its ease-of-use.

Stop the container

To shut down the container, in the command-line environment where the container is running, select `Ctrl+C`.

Troubleshooting

If you run the container with an output mount and logging enabled, the container generates log files that are helpful to troubleshoot issues that happen while starting or running the container.

> **TIP**
> For more troubleshooting information and guidance, see Cognitive Services containers frequently asked questions (FAQ).

Billing

The Cognitive Services containers send billing information to Azure, using the corresponding resource on your Azure account.

Queries to the container are billed at the pricing tier of the Azure resource that's used for the `ApiKey`.

Azure Cognitive Services containers aren't licensed to run without being connected to the metering / billing endpoint. You must enable the containers to communicate billing information with the billing endpoint at all times. Cognitive Services containers don't send customer data, such as the image or text that's being analyzed, to Microsoft.

Connect to Azure

The container needs the billing argument values to run. These values allow the container to connect to the billing endpoint. The container reports usage about every 10 to 15 minutes. If the container doesn't connect to Azure within the allowed time window, the container continues to run but doesn't serve queries until the billing endpoint is restored. The connection is attempted 10 times at the same time interval of 10 to 15 minutes. If it can't connect to the billing endpoint within the 10 tries, the container stops serving requests. See the Cognitive Services container FAQ for an example of the information sent to Microsoft for billing.

Billing arguments

The `docker run` command will start the container when all three of the following options are provided with valid values:

OPTION	DESCRIPTION
`ApiKey`	The API key of the Cognitive Services resource that's used to track billing information. The value of this option must be set to an API key for the provisioned resource that's specified in `Billing`.

OPTION	DESCRIPTION
`Billing`	The endpoint of the Cognitive Services resource that's used to track billing information. The value of this option must be set to the endpoint URI of a provisioned Azure resource.
`Eula`	Indicates that you accepted the license for the container. The value of this option must be set to **accept**.

For more information about these options, see Configure containers.

Summary

In this article, you learned concepts and workflow for downloading, installing, and running Computer Vision containers. In summary:

- Computer Vision provides a Linux container for Docker, encapsulating Read.
- The read container image requires an application to run it.
- Container images run in Docker.
- You can use either the REST API or SDK to call operations in Read OCR containers by specifying the host URI of the container.
- You must specify billing information when instantiating a container.

> **IMPORTANT**
> Cognitive Services containers are not licensed to run without being connected to Azure for metering. Customers need to enable the containers to communicate billing information with the metering service at all times. Cognitive Services containers do not send customer data (for example, the image or text that is being analyzed) to Microsoft.

Next steps

- Review Configure containers for configuration settings
- Review the OCR overview to learn more about recognizing printed and handwritten text
- Refer to the Read API for details about the methods supported by the container.
- Refer to Frequently asked questions (FAQ) to resolve issues related to Computer Vision functionality.
- Use more Cognitive Services Containers

Configure Read OCR Docker containers

4/12/2021 • 9 minutes to read • Edit Online

You configure the Computer Vision Read OCR container's runtime environment by using the `docker run` command arguments. This container has several required settings, along with a few optional settings. Several examples of the command are available. The container-specific settings are the billing settings.

Configuration settings

The container has the following configuration settings:

REQUIRED	SETTING	PURPOSE
Yes	ApiKey	Tracks billing information.
No	ApplicationInsights	Enables adding Azure Application Insights telemetry support to your container.
Yes	Billing	Specifies the endpoint URI of the service resource on Azure.
Yes	Eula	Indicates that you've accepted the license for the container.
No	Fluentd	Writes log and, optionally, metric data to a Fluentd server.
No	HTTP Proxy	Configures an HTTP proxy for making outbound requests.
No	Logging	Provides ASP.NET Core logging support for your container.
No	Mounts	Reads and writes data from the host computer to the container and from the container back to the host computer.

IMPORTANT

The `ApiKey`, `Billing`, and `Eula` settings are used together, and you must provide valid values for all three of them; otherwise your container won't start. For more information about using these configuration settings to instantiate a container, see Billing.

The container also has the following container-specific configuration settings:

REQUIRED	SETTING	PURPOSE
No	ReadEngineConfig:ResultExpirationPeriod	v2.0 containers only. Result expiration period in hours. The default is 48 hours. The setting specifies when the system should clear recognition results. For example, if `resultExpirationPeriod=1`, the system clears the recognition result 1 hour after the process. If `resultExpirationPeriod=0`, the system clears the recognition result after the result is retrieved.
No	Cache:Redis	v2.0 containers only. Enables Redis storage for storing results. A cache is *required* if multiple read OCR containers are placed behind a load balancer.

REQUIRED	SETTING	PURPOSE
No	Queue:RabbitMQ	v2.0 containers only. Enables RabbitMQ for dispatching tasks. The setting is useful when multiple read OCR containers are placed behind a load balancer.
No	Queue:Azure:QueueVisibilityTimeoutInMilliseconds	v3.x containers only. The time for a message to be invisible when another worker is processing it.
No	Storage::DocumentStore::MongoDB	v2.0 containers only. Enables MongoDB for permanent result storage.
No	Storage:ObjectStore:AzureBlob:ConnectionString	v3.x containers only. Azure blob storage connection string.
No	Storage:TimeToLiveInDays	v3.x containers only. Result expiration period in days. The setting specifies when the system should clear recognition results. The default is 2 days (48 hours), which means any result live for longer than that period is not guaranteed to be successfully retrieved.
No	Task:MaxRunningTimeSpanInMinutes	v3.x containers only. Maximum running time for a single request. The default is 60 minutes.

ApiKey configuration setting

The `ApiKey` setting specifies the Azure `Cognitive Services` resource key used to track billing information for the container. You must specify a value for the ApiKey and the value must be a valid key for the *Cognitive Services* resource specified for the `Billing` configuration setting.

This setting can be found in the following place:

- Azure portal: **Cognitive Services** Resource Management, under **Keys**

ApplicationInsights setting

The `ApplicationInsights` setting allows you to add Azure Application Insights telemetry support to your container. Application Insights provides in-depth monitoring of your container. You can easily monitor your container for availability, performance, and usage. You can also quickly identify and diagnose errors in your container.

The following table describes the configuration settings supported under the `ApplicationInsights` section.

REQUIRED	NAME	DATA TYPE	DESCRIPTION
No	`InstrumentationKey`	String	The instrumentation key of the Application Insights instance to which telemetry data for the container is sent. For more information, see Application Insights for ASP.NET Core. Example: `InstrumentationKey=123456789`

Billing configuration setting

The `Billing` setting specifies the endpoint URI of the *Cognitive Services* resource on Azure used to meter billing information for the container. You must specify a value for this configuration setting, and the value must be a valid endpoint URI for a *Cognitive Services* resource on Azure. The container reports usage about every 10 to 15 minutes.

This setting can be found in the following place:

- Azure portal: **Cognitive Services** Overview, labeled `Endpoint`

Remember to add the `vision/<version>` routing to the endpoint URI as shown in the following table.

REQUIRED	NAME	DATA TYPE	DESCRIPTION
Yes	`Billing`	String	Billing endpoint URI Example: `Billing=https://westcentralus.api.cognitive.microso`

Eula setting

The `Eula` setting indicates that you've accepted the license for the container. You must specify a value for this configuration setting, and the value must be set to `accept`.

REQUIRED	NAME	DATA TYPE	DESCRIPTION
Yes	`Eula`	String	License acceptance Example: `Eula=accept`

Cognitive Services containers are licensed under your agreement governing your use of Azure. If you do not have an existing agreement governing your use of Azure, you agree that your agreement governing use of Azure is the Microsoft Online Subscription Agreement, which incorporates the Online Services Terms. For previews, you also agree to the Supplemental Terms of Use for Microsoft Azure Previews. By using the container you agree to these terms.

Fluentd settings

Fluentd is an open-source data collector for unified logging. The `Fluentd` settings manage the container's connection to a Fluentd server. The container includes a Fluentd logging provider, which allows your container to write logs and, optionally, metric data to a Fluentd server.

The following table describes the configuration settings supported under the `Fluentd` section.

NAME	DATA TYPE	DESCRIPTION
`Host`	String	The IP address or DNS host name of the Fluentd server.
`Port`	Integer	The port of the Fluentd server. The default value is 24224.
`HeartbeatMs`	Integer	The heartbeat interval, in milliseconds. If no event traffic has been sent before this interval expires, a heartbeat is sent to the Fluentd server. The default value is 60000 milliseconds (1 minute).
`SendBufferSize`	Integer	The network buffer space, in bytes, allocated for send operations. The default value is 32768 bytes (32 kilobytes).
`TlsConnectionEstablishmentTimeoutMs`	Integer	The timeout, in milliseconds, to establish a SSL/TLS connection with the Fluentd server. The default value is 10000 milliseconds (10 seconds). If `UseTLS` is set to false, this value is ignored.
`UseTLS`	Boolean	Indicates whether the container should use SSL/TLS for communicating with the Fluentd server. The default value is false.

HTTP proxy credentials settings

If you need to configure an HTTP proxy for making outbound requests, use these two arguments:

NAME	DATA TYPE	DESCRIPTION
HTTP_PROXY	string	The proxy to use, for example, `http://proxy:8888` `<proxy-url>`
HTTP_PROXY_CREDS	string	Any credentials needed to authenticate against the proxy, for example, `username:password`. This value **must be in lower-case**.
`<proxy-user>`	string	The user for the proxy.
`<proxy-password>`	string	The password associated with `<proxy-user>` for the proxy.

```
docker run --rm -it -p 5000:5000 \
--memory 2g --cpus 1 \
--mount type=bind,src=/home/azureuser/output,target=/output \
<registry-location>/<image-name> \
Eula=accept \
Billing=<endpoint> \
ApiKey=<api-key> \
HTTP_PROXY=<proxy-url> \
HTTP_PROXY_CREDS=<proxy-user>:<proxy-password> \
```

Logging settings

The `Logging` settings manage ASP.NET Core logging support for your container. You can use the same configuration settings and values for your container that you use for an ASP.NET Core application.

The following logging providers are supported by the container:

PROVIDER	PURPOSE
Console	The ASP.NET Core `Console` logging provider. All of the ASP.NET Core configuration settings and default values for this logging provider are supported.
Debug	The ASP.NET Core `Debug` logging provider. All of the ASP.NET Core configuration settings and default values for this logging provider are supported.
Disk	The JSON logging provider. This logging provider writes log data to the output mount.

This container command stores logging information in the JSON format to the output mount:

```
docker run --rm -it -p 5000:5000 \
--memory 2g --cpus 1 \
--mount type=bind,src=/home/azureuser/output,target=/output \
<registry-location>/<image-name> \
Eula=accept \
Billing=<endpoint> \
ApiKey=<api-key> \
Logging:Disk:Format=json
```

This container command shows debugging information, prefixed with `dbug`, while the container is running:

```
docker run --rm -it -p 5000:5000 \
--memory 2g --cpus 1 \
<registry-location>/<image-name> \
Eula=accept \
Billing=<endpoint> \
ApiKey=<api-key> \
Logging:Console:LogLevel:Default=Debug
```

Disk logging

The `Disk` logging provider supports the following configuration settings:

NAME	DATA TYPE	DESCRIPTION
`Format`	String	The output format for log files. **Note:** This value must be set to `json` to enable the logging provider. If this value is specified without also specifying an output mount while instantiating a container, an error occurs.
`MaxFileSize`	Integer	The maximum size, in megabytes (MB), of a log file. When the size of the current log file meets or exceeds this value, a new log file is started by the logging provider. If -1 is specified, the size of the log file is limited only by the maximum file size, if any, for the output mount. The default value is 1.

For more information about configuring ASP.NET Core logging support, see Settings file configuration.

Mount settings

Use bind mounts to read and write data to and from the container. You can specify an input mount or output mount by specifying the `--mount` option in the docker run command.

The Computer Vision containers don't use input or output mounts to store training or service data.

The exact syntax of the host mount location varies depending on the host operating system. Additionally, the host computer's mount location may not be accessible due to a conflict between permissions used by the Docker service account and the host mount location permissions.

OPTIONAL	NAME	DATA TYPE	DESCRIPTION
Not allowed	`Input`	String	Computer Vision containers do not use this.
Optional	`Output`	String	The target of the output mount. The default value is `/output`. This is the location of the logs. This includes container logs. Example: `--mount type=bind,src=c:\output,target=/output`

Example docker run commands

The following examples use the configuration settings to illustrate how to write and use `docker run` commands. Once running, the container continues to run until you stop it.

- **Line-continuation character**: The Docker commands in the following sections use the back slash, `\`, as a line continuation character. Replace or remove this based on your host operating system's requirements.
- **Argument order**: Do not change the order of the arguments unless you are very familiar with Docker containers.

Replace {*argument_name*} with your own values:

PLACEHOLDER	VALUE	FORMAT OR EXAMPLE
{API_KEY}	The endpoint key of the `Computer Vision` resource on the Azure `Computer Vision` Keys page.	`xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`
{ENDPOINT_URI}	The billing endpoint value is available on the Azure `Computer Vision` Overview page.	See gathering required parameters for explicit examples.

> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

IMPORTANT

The `Eula`, `Billing`, and `ApiKey` options must be specified to run the container; otherwise, the container won't start. For more information, see Billing. The ApiKey value is the **Key** from the Azure `Cognitive Services` Resource keys page.

Container Docker examples

The following Docker examples are for the Read OCR container.

- Version 3.2
- Version 2.0-preview

Basic example

```
docker run --rm -it -p 5000:5000 --memory 18g --cpus 8 \
mcr.microsoft.com/azure-cognitive-services/vision/read:3.2 \
Eula=accept \
Billing={ENDPOINT_URI} \
ApiKey={API_KEY}
```

Logging example

```
docker run --rm -it -p 5000:5000 --memory 18g --cpus 8 \
mcr.microsoft.com/azure-cognitive-services/vision/read:3.2 \
Eula=accept \
Billing={ENDPOINT_URI} \
ApiKey={API_KEY}
Logging:Console:LogLevel:Default=Information
```

Next steps

- Review How to install and run containers.

Migrate to the Read v3.x OCR containers

5/25/2021 • 2 minutes to read • Edit Online

If you're using version 2 of the Computer Vision Read OCR container, Use this article to learn about upgrading your application to use version 3.x of the container.

Configuration changes

- `ReadEngineConfig:ResultExpirationPeriod` is no longer supported. The Read OCR container has a built Cron job that removes the results and metadata associated with a request after 48 hours.
- `Cache:Redis:Configuration` is no longer supported. The Cache is not used in the v3.x containers, so you don't need to set it.

API changes

The Read v3.2 container uses version 3 of the Computer Vision API and has the following endpoints:

- `/vision/v3.2-preview.1/read/analyzeResults/{operationId}`
- `/vision/v3.2-preview.1/read/analyze`
- `/vision/v3.2-preview.1/read/syncAnalyze`

See the Computer Vision v3 REST API migration guide for detailed information on updating your applications to use version 3 of cloud-based Read API. This information applies to the container as well. Sync operations are only supported in containers.

Memory requirements

The requirements and recommendations are based on benchmarks with a single request per second, using an 8-MB image of a scanned business letter that contains 29 lines and a total of 803 characters. The following table describes the minimum and recommended allocation of resources for each Read OCR container.

CONTAINER	MINIMUM	RECOMMENDED
Read 3.2-preview	8 cores, 16-GB memory	8 cores, 24-GB memory

Each core must be at least 2.6 gigahertz (GHz) or faster.

Core and memory correspond to the `--cpus` and `--memory` settings, which are used as part of the docker run command.

Storage implementations

> **NOTE**
> MongoDB is no longer supported in 3.x versions of the container. Instead, the containers support Azure Storage and offline file systems.

IMPLEMENTATION	REQUIRED RUNTIME ARGUMENT(S)
File level (default)	No runtime arguments required. `/share` directory will be used.
Azure Blob	`Storage:ObjectStore:AzureBlob:ConnectionString={AzureStorageConnectionString}`

Queue implementations

In v3.x of the container, RabbitMQ is currently not supported. The supported backing implementations are:

IMPLEMENTATION	RUNTIME ARGUMENT(S)	INTENDED USE
In Memory (default)	No runtime arguments required.	Development and testing
Azure Queues	`Queue:Azure:ConnectionString={AzureStorageConnectionString}`	Production
RabbitMQ	Unavailable	Production

For added redundancy, the Read v3.x container uses a visibility timer to ensure requests can be successfully processed if a crash occurs when running in a multi-container setup.

Set the timer with `Queue:Azure:QueueVisibilityTimeoutInMilliseconds`, which sets the time for a message to be invisible when another worker is processing it. To avoid pages from being redundantly processed, we recommend setting the timeout period to 120 seconds. The default value is 30 seconds.

DEFAULT VALUE	RECOMMENDED VALUE
30000	120000

Next steps

- Review Configure containers for configuration settings
- Review OCR overview to learn more about recognizing printed and handwritten text
- Refer to the Read API for details about the methods supported by the container.
- Refer to Frequently asked questions (FAQ) to resolve issues related to Computer Vision functionality.
- Use more Cognitive Services Containers

Use Computer Vision container with Kubernetes and Helm

4/3/2021 • 10 minutes to read • Edit Online

One option to manage your Computer Vision containers on-premises is to use Kubernetes and Helm. Using Kubernetes and Helm to define a Computer Vision container image, we'll create a Kubernetes package. This package will be deployed to a Kubernetes cluster on-premises. Finally, we'll explore how to test the deployed services. For more information about running Docker containers without Kubernetes orchestration, see install and run Computer Vision containers.

Prerequisites

The following prerequisites before using Computer Vision containers on-premises:

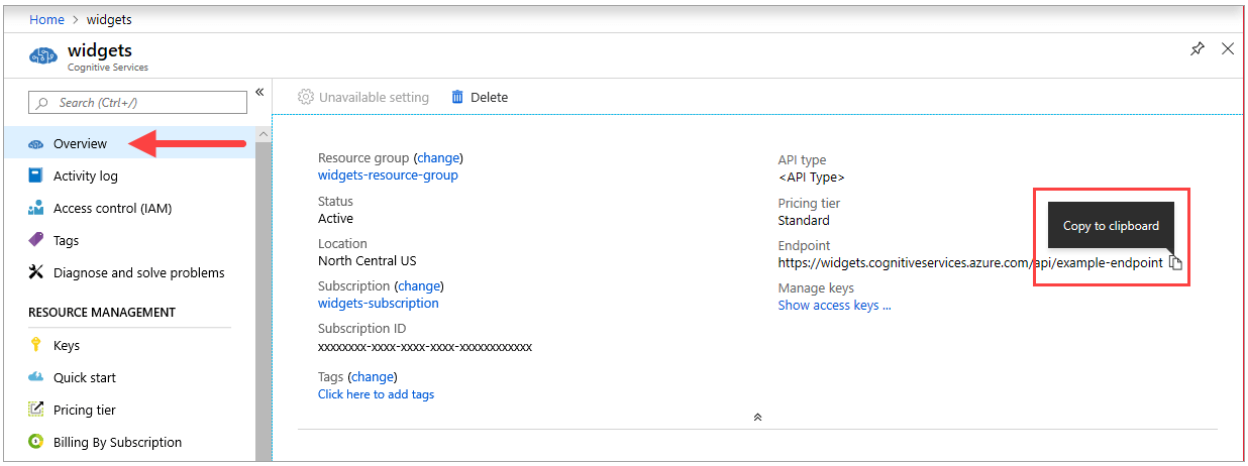
REQUIRED	PURPOSE
Azure Account	If you don't have an Azure subscription, create a free account before you begin.
Kubernetes CLI	The Kubernetes CLI is required for managing the shared credentials from the container registry. Kubernetes is also needed before Helm, which is the Kubernetes package manager.
Helm CLI	Install the Helm CLI, which is used to to install a helm chart (container package definition).
Computer Vision resource	In order to use the container, you must have: An Azure **Computer Vision** resource and the associated API key the endpoint URI. Both values are available on the Overview and Keys pages for the resource and are required to start the container. **{API_KEY}**: One of the two available resource keys on the **Keys** page **{ENDPOINT_URI}**: The endpoint as provided on the **Overview** page

Gathering required parameters

There are three primary parameters for all Cognitive Services' containers that are required. The end-user license agreement (EULA) must be present with a value of `accept`. Additionally, both an Endpoint URL and API Key are needed.

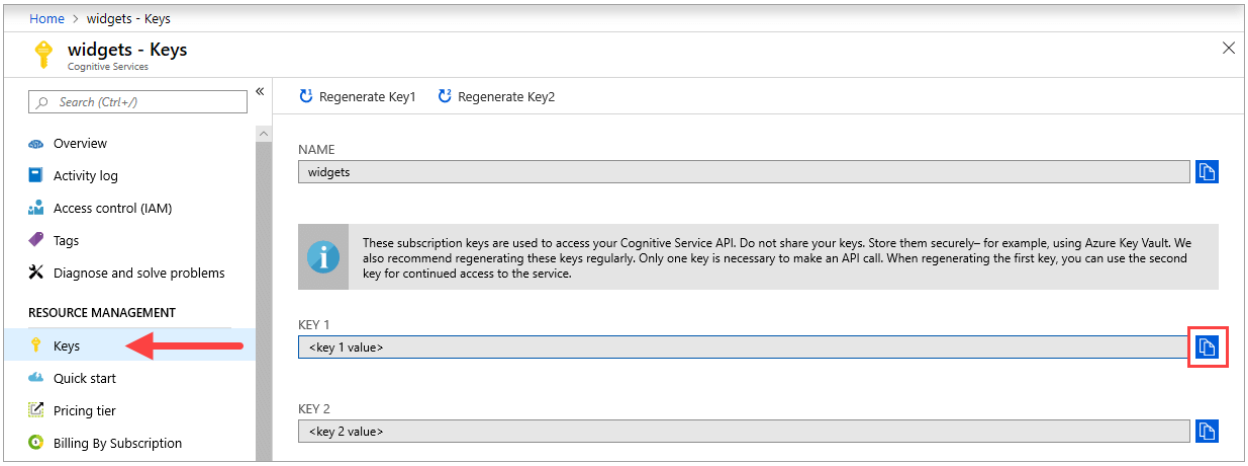
Endpoint URI `{ENDPOINT_URI}`

The **Endpoint** URI value is available on the Azure portal *Overview* page of the corresponding Cognitive Service resource. Navigate to the *Overview* page, hover over the Endpoint, and a `Copy to clipboard` icon will appear. Copy and use where needed.



Keys `{API_KEY}`

This key is used to start the container, and is available on the Azure portal's Keys page of the corresponding Cognitive Service resource. Navigate to the *Keys* page, and click on the `Copy to clipboard` icon.



> **IMPORTANT**
> These subscription keys are used to access your Cognitive Service API. Do not share your keys. Store them securely, for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

The host computer

The host is a x64-based computer that runs the Docker container. It can be a computer on your premises or a Docker hosting service in Azure, such as:

- Azure Kubernetes Service.
- Azure Container Instances.
- A Kubernetes cluster deployed to Azure Stack. For more information, see Deploy Kubernetes to Azure Stack.

Container requirements and recommendations

> **NOTE**
> The requirements and recommendations are based on benchmarks with a single request per second, using an 8-MB image of a scanned business letter that contains 29 lines and a total of 803 characters.

The following table describes the minimum and recommended allocation of resources for each Read OCR container.

CONTAINER	MINIMUM	RECOMMENDED
Read 2.0-preview	1 core, 8-GB memory	8 cores, 16-GB memory
Read 3.2	8 cores, 16-GB memory	8 cores, 24-GB memory

- Each core must be at least 2.6 gigahertz (GHz) or faster.

Core and memory correspond to the `--cpus` and `--memory` settings, which are used as part of the `docker run` command.

Connect to the Kubernetes cluster

The host computer is expected to have an available Kubernetes cluster. See this tutorial on deploying a Kubernetes cluster for a conceptual understanding of how to deploy a Kubernetes cluster to a host computer. You can find more information on deployments in the Kubernetes documentation.

Configure Helm chart values for deployment

Begin by creating a folder named *read*. Then, paste the following YAML content in a new file named `chart.yaml`:

```
apiVersion: v2
name: read
version: 1.0.0
description: A Helm chart to deploy the Read OCR container to a Kubernetes cluster
dependencies:
- name: rabbitmq
  condition: read.image.args.rabbitmq.enabled
  version: ^6.12.0
  repository: https://kubernetes-charts.storage.googleapis.com/
- name: redis
  condition: read.image.args.redis.enabled
  version: ^6.0.0
  repository: https://kubernetes-charts.storage.googleapis.com/
```

To configure the Helm chart default values, copy and paste the following YAML into a file named `values.yaml`. Replace the `# {ENDPOINT_URI}` and `# {API_KEY}` comments with your own values. Configure resultExpirationPeriod, Redis, and RabbitMQ if needed.

```
# These settings are deployment specific and users can provide customizations
read:
  enabled: true
  image:
    name: cognitive-services-read
    registry:  mcr.microsoft.com/
    repository: azure-cognitive-services/vision/read
    tag: 3.2-preview.1
    args:
      eula: accept
      billing: # {ENDPOINT_URI}
      apikey: # {API_KEY}

      # Result expiration period setting. Specify when the system should clean up recognition results.
      # For example, resultExpirationPeriod=1, the system will clear the recognition result 1hr after the
process.
      # resultExpirationPeriod=0, the system will clear the recognition result after result retrieval.
      resultExpirationPeriod: 1

      # Redis storage, if configured, will be used by read OCR container to store result records.
      # A cache is required if multiple read OCR containers are placed behind load balancer.
      redis:
        enabled: false # {true/false}
        password: password

      # RabbitMQ is used for dispatching tasks. This can be useful when multiple read OCR containers are
      # placed behind load balancer.
      rabbitmq:
        enabled: false # {true/false}
        rabbitmq:
          username: user
          password: password
```

IMPORTANT

- If the `billing` and `apikey` values aren't provided, the services expire after 15 minutes. Likewise, verification fails because the services aren't available.
- If you deploy multiple Read OCR containers behind a load balancer, for example, under Docker Compose or Kubernetes, you must have an external cache. Because the processing container and the GET request container might not be the same, an external cache stores the results and shares them across containers. For details about cache settings, see Configure Computer Vision Docker containers.

Create a *templates* folder under the *read* directory. Copy and paste the following YAML into a file named `deployment.yaml`. The `deployment.yaml` file will serve as a Helm template.

> Templates generate manifest files, which are YAML-formatted resource descriptions that Kubernetes can understand. - Helm Chart Template Guide

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: read
  labels:
    app: read-deployment
spec:
  selector:
    matchLabels:
      app: read-app
  template:
    metadata:
      labels:
        app: read-app
    spec:
      containers:
      - name: {{.Values.read.image.name}}
        image: {{.Values.read.image.registry}}{{.Values.read.image.repository}}
        ports:
        - containerPort: 5000
        env:
        - name: EULA
          value: {{.Values.read.image.args.eula}}
        - name: billing
          value: {{.Values.read.image.args.billing}}
        - name: apikey
          value: {{.Values.read.image.args.apikey}}
        args:
        - ReadEngineConfig:ResultExpirationPeriod={{ .Values.read.image.args.resultExpirationPeriod }}
        {{- if .Values.read.image.args.rabbitmq.enabled }}
        - Queue:RabbitMQ:HostName={{ include "rabbitmq.hostname" . }}
        - Queue:RabbitMQ:Username={{ .Values.read.image.args.rabbitmq.rabbitmq.username }}
        - Queue:RabbitMQ:Password={{ .Values.read.image.args.rabbitmq.rabbitmq.password }}
        {{- end }}
        {{- if .Values.read.image.args.redis.enabled }}
        - Cache:Redis:Configuration={{ include "redis.connStr" . }}
        {{- end }}
      imagePullSecrets:
      - name: {{.Values.read.image.pullSecret}}
---
apiVersion: v1
kind: Service
metadata:
  name: read-service
spec:
  type: LoadBalancer
  ports:
  - port: 5000
  selector:
    app: read-app
```

In the same *templates* folder, copy and paste the following helper functions into `helpers.tpl`. `helpers.tpl` defines useful functions to help generate Helm template.

> **NOTE**
>
> This article contains references to the term slave, a term that Microsoft no longer uses. When the term is removed from the software, we'll remove it from this article.

```
{{- define "rabbitmq.hostname" -}}
{{- printf "%s-rabbitmq" .Release.Name -}}
{{- end -}}

{{- define "redis.connStr" -}}
{{- $hostMain := printf "%s-redis-master:6379" .Release.Name }}
{{- $hostReplica := printf "%s-redis-slave:6379" .Release.Name -}}
{{- $passWord := printf "password=%s" .Values.read.image.args.redis.password -}}
{{- $connTail := "ssl=False,abortConnect=False" -}}
{{- printf "%s,%s,%s,%s" $hostMain $hostReplica $passWord $connTail -}}
{{- end -}}
```

The template specifies a load balancer service and the deployment of your container/image for Read.

The Kubernetes package (Helm chart)

The *Helm chart* contains the configuration of which docker image(s) to pull from the `mcr.microsoft.com` container registry.

> A Helm chart is a collection of files that describe a related set of Kubernetes resources. A single chart might be used to deploy something simple, like a memcached pod, or something complex, like a full web app stack with HTTP servers, databases, caches, and so on.

The provided *Helm charts* pull the docker images of the Computer Vision Service, and the corresponding service from the `mcr.microsoft.com` container registry.

Install the Helm chart on the Kubernetes cluster

To install the *helm chart*, we'll need to execute the `helm install` command. Ensure to execute the install command from the directory above the `read` folder.

```
helm install read ./read
```

Here is an example output you might expect to see from a successful install execution:

```
NAME: read
LAST DEPLOYED: Thu Sep 04 13:24:06 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME                    READY  STATUS             RESTARTS  AGE
read-57cb76bcf7-45sdh   0/1    ContainerCreating  0         0s

==> v1/Service
NAME     TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)         AGE
read     LoadBalancer  10.110.44.86  localhost    5000:31301/TCP  0s

==> v1beta1/Deployment
NAME    READY  UP-TO-DATE  AVAILABLE  AGE
read    0/1    1           0          0s
```

The Kubernetes deployment can take over several minutes to complete. To confirm that both pods and services are properly deployed and available, execute the following command:

```
kubectl get all
```

You should expect to see something similar to the following output:

```
kubectl get all
NAME                         READY   STATUS    RESTARTS   AGE
pod/read-57cb76bcf7-45sdh    1/1     Running   0          17s

NAME                 TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
service/kubernetes   ClusterIP      10.96.0.1      <none>        443/TCP          45h
service/read         LoadBalancer   10.110.44.86   localhost     5000:31301/TCP   17s

NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/read 1/1     1            1           17s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/read-57cb76bcf7    1         1         1       17s
```

Deploy multiple v3 containers on the Kubernetes cluster

Starting in v3 of the container, you can use the containers in parallel on both a task and page level.

By design, each v3 container has a dispatcher and a recognition worker. The dispatcher is responsible for splitting a multi-page task into multiple single page sub-tasks. The recognition worker is optimized for recognizing a single page document. To achieve page level parallelism, deploy multiple v3 containers behind a load balancer and let the containers share a universal storage and queue.

> **NOTE**
> Currently only Azure Storage and Azure Queue are supported.

The container receiving the request can split the task into single page sub-tasks, and add them to the universal queue. Any recognition worker from a less busy container can consume single page sub-tasks from the queue, perform recognition, and upload the result to the storage. The throughput can be improved up to `n` times, depending on the number of containers that are deployed.

The v3 container exposes the liveness probe API under the `/ContainerLiveness` path. Use the following deployment example to configure a liveness probe for Kubernetes.

Copy and paste the following YAML into a file named `deployment.yaml`. Replace the `# {ENDPOINT_URI}` and `# {API_KEY}` comments with your own values. Replace the `# {AZURE_STORAGE_CONNECTION_STRING}` comment with your Azure Storage Connection String. Configure `replicas` to the number you want, which is set to `3` in the following example.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: read
  labels:
    app: read-deployment
spec:
  selector:
    matchLabels:
      app: read-app
  replicas: # {NUMBER_OF_READ_CONTAINERS}
  template:
    metadata:
      labels:
        app: read-app
    spec:
      containers:
      - name: cognitive-services-read
        image: mcr.microsoft.com/azure-cognitive-services/vision/read
        ports:
        - containerPort: 5000
        env:
        - name: EULA
          value: accept
        - name: billing
          value: # {ENDPOINT_URI}
        - name: apikey
          value: # {API_KEY}
        - name: Storage__ObjectStore__AzureBlob__ConnectionString
          value: # {AZURE_STORAGE_CONNECTION_STRING}
        - name: Queue__Azure__ConnectionString
          value: # {AZURE_STORAGE_CONNECTION_STRING}
        livenessProbe:
          httpGet:
            path: /ContainerLiveness
            port: 5000
          initialDelaySeconds: 60
          periodSeconds: 60
          timeoutSeconds: 20
---
apiVersion: v1
kind: Service
metadata:
  name: azure-cognitive-service-read
spec:
  type: LoadBalancer
  ports:
  - port: 5000
    targetPort: 5000
  selector:
    app: read-app
```

Run the following command.

```
kubectl apply -f deployment.yaml
```

Below is an example output you might see from a successful deployment execution:

```
deployment.apps/read created
service/azure-cognitive-service-read created
```

The Kubernetes deployment can take several minutes to complete. To confirm that both pods and services are properly deployed and available, then execute the following command:

```
kubectl get all
```

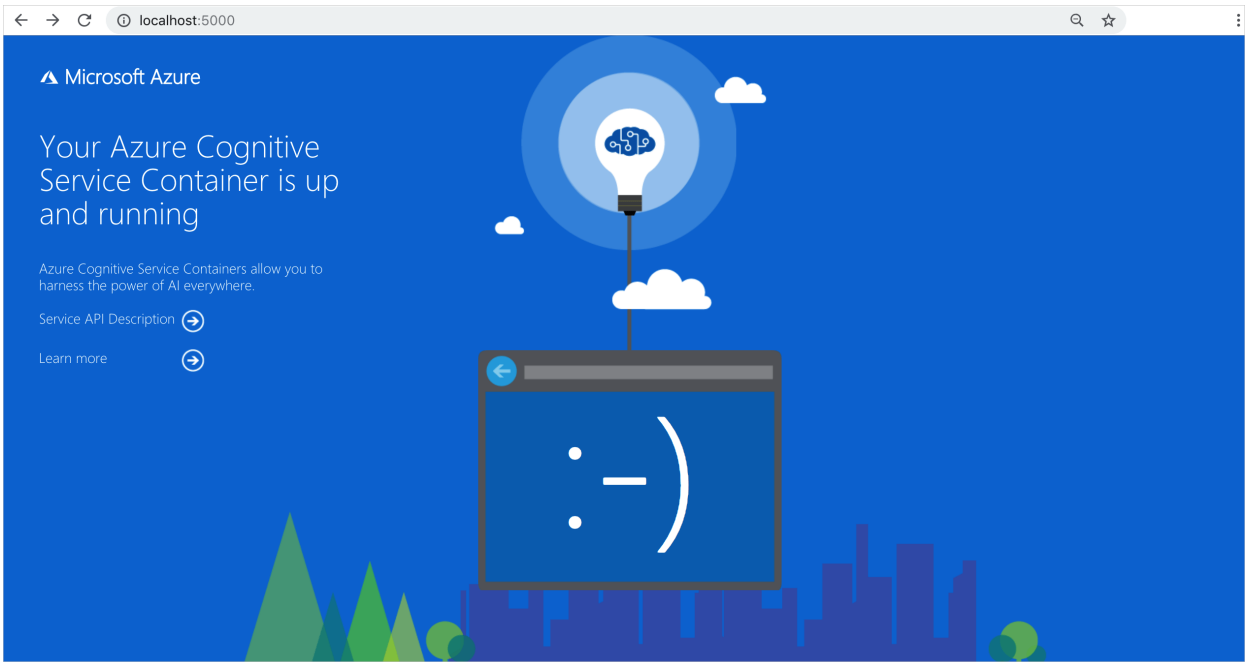
You should see console output similar to the following:

```
kubectl get all
NAME                         READY   STATUS    RESTARTS   AGE
pod/read-6cbbb6678-58s9t   1/1     Running   0          3s
pod/read-6cbbb6678-kz7v4   1/1     Running   0          3s
pod/read-6cbbb6678-s2pct   1/1     Running   0          3s

NAME                                    TYPE           CLUSTER-IP   EXTERNAL-IP    PORT(S)          AGE
service/azure-cognitive-service-read    LoadBalancer   10.0.134.0   <none>         5000:30846/TCP   17h
service/kubernetes                      ClusterIP      10.0.0.1     <none>         443/TCP          78d

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/read    3/3     3            3           3s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/read-6cbbb6678   3         3         3       3s
```

Validate that a container is running

There are several ways to validate that the container is running. Locate the *External IP* address and exposed port of the container in question, and open your favorite web browser. Use the various request URLs below to validate the container is running. The example request URLs listed below are `http://localhost:5000`, but your specific container may vary. Keep in mind that you're to rely on your container's *External IP* address and exposed port.

REQUEST URL	PURPOSE
`http://localhost:5000/`	The container provides a home page.
`http://localhost:5000/ready`	Requested with GET, this provides a verification that the container is ready to accept a query against the model. This request can be used for Kubernetes liveness and readiness probes.
`http://localhost:5000/status`	Also requested with GET, this verifies if the api-key used to start the container is valid without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes.
`http://localhost:5000/swagger`	The container provides a full set of documentation for the endpoints and a **Try it out** feature. With this feature, you can enter your settings into a web-based HTML form and make the query without having to write any code. After the query returns, an example CURL command is provided to demonstrate the HTTP headers and body format that's required.



Next steps

For more details on installing applications with Helm in Azure Kubernetes Service (AKS), visit here.

Cognitive Services Containers

Deploy and run container on Azure Container Instance

4/29/2021 • 7 minutes to read • Edit Online

With the following steps, scale Azure Cognitive Services applications in the cloud easily with Azure Container Instances. Containerization helps you focus on building your applications instead of managing the infrastructure. For more information on using containers, see features and benefits.

Prerequisites

The recipe works with any Cognitive Services container. The Cognitive Service resource must be created before using the recipe. Each Cognitive Service that supports containers has a "How to install" article for installing and configuring the service for a container. Some services require a file or set of files as input for the container, it is important that you understand and have used the container successfully before using this solution.

- An Azure resource for the Azure Cognitive Service you're using.
- Cognitive Service **endpoint URL** - review your specific service's "How to install" for the container, to find where the endpoint URL is from within the Azure portal, and what a correct example of the URL looks like. The exact format can change from service to service.
- Cognitive Service **key** - the keys are on the **Keys** page for the Azure resource. You only need one of the two keys. The key is a string of 32 alpha-numeric characters.
- A single Cognitive Services Container on your local host (your computer). Make sure you can:
  - Pull down the image with a `docker pull` command.
  - Run the local container successfully with all required configuration settings with a `docker run` command.
  - Call the container's endpoint, getting a response of HTTP 2xx and a JSON response back.

All variables in angle brackets, `<>`, need to be replaced with your own values. This replacement includes the angle brackets.

> **IMPORTANT**
> The LUIS container requires a `.gz` model file that is pulled in at runtime. The container must be able to access this model file via a volume mount from the container instance. To upload a model file, follow these steps:
>
> 1. Create an Azure file share. Take note of the Azure Storage account name, key, and file share name as you'll need them later.
> 2. export your LUIS model (packaged app) from the LUIS portal.
> 3. In the Azure portal, navigate to the **Overview** page of your storage account resource, and select **File shares**.
> 4. Select the file share name that you recently created, then select **Upload**. Then upload your packaged app.

- Azure portal
- CLI

Create an Azure Container Instance resource using the Azure portal

1. Go to the Create page for Container Instances.
2. On the **Basics** tab, enter the following details:

SETTING	VALUE
Subscription	Select your subscription.
Resource group	Select the available resource group or create a new one such as `cognitive-services`.
Container name	Enter a name such as `cognitive-container-instance`. The name must be in lower caps.
Location	Select a region for deployment.
Image type	If your container image is stored in a container registry that doesn't require credentials, choose `Public`. If accessing your container image requires credentials, choose `Private`. Refer to container repositories and images for details on whether or not the container image is `Public` or `Private` ("Public Preview").
Image name	Enter the Cognitive Services container location. The location is what's used as an argument to the `docker pull` command. Refer to the container repositories and images for the available image names and their corresponding repository. The image name must be fully qualified specifying three parts. First, the container registry, then the repository, finally the image name: `<container-registry>/<repository>/<image-name>`. Here is an example, `mcr.microsoft.com/azure-cognitive-services/keyphrase` would represent the Key Phrase Extraction image in the Microsoft Container Registry under the Azure Cognitive Services repository. Another example is, `containerpreview.azurecr.io/microsoft/cognitive-services-speech-to-text` which would represent the Speech to Text image in the Microsoft repository of the Container Preview container registry.
OS type	`Linux`
Size	Change size to the suggested recommendations for your specific Cognitive Service container: 2 CPU cores 4 GB

3. On the **Networking** tab, enter the following details:

SETTING	VALUE

SETTING	VALUE
Ports	Set the TCP port to `5000`. Exposes the container on port 5000.

4. On the **Advanced** tab, enter the required **Environment Variables** for the container billing settings of the Azure Container Instance resource:

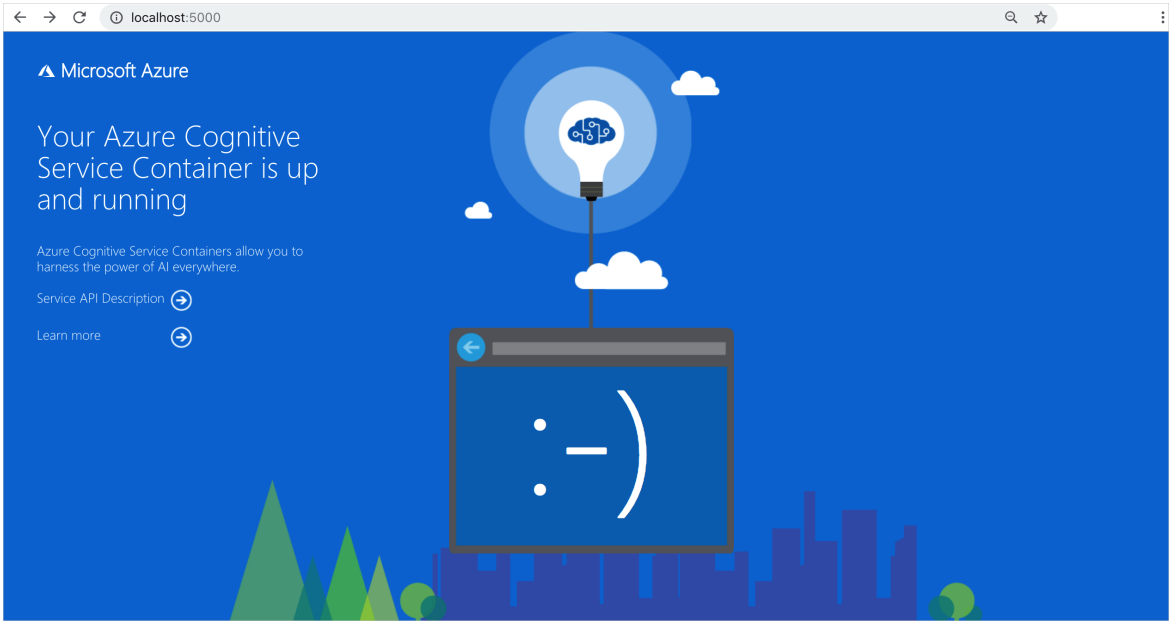
KEY	VALUE
`ApiKey`	Copied from the **Keys and endpoint** page of the resource. It is a 32 alphanumeric-character string with no spaces or dashes, `xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`.
`Billing`	Your endpoint URL copied from the **Keys and endpoint** page of the resource.
`Eula`	`accept`

5. Click **Review and Create**
6. After validation passes, click **Create** to finish the creation process
7. When the resource is successfully deployed, it's ready

Use the Container Instance

- Azure portal
- CLI

1. Select the **Overview** and copy the IP address. It will be a numeric IP address such as `55.55.55.55`.
2. Open a new browser tab and use the IP address, for example, `http://<IP-address>:5000 (http://55.55.55.55:5000)`. You will see the container's home page, letting you know the container is running.



3. Select **Service API Description** to view the swagger page for the container.
4. Select any of the **POST** APIs and select **Try it out**. The parameters are displayed including the input. Fill

in the parameters.

5. Select **Execute** to send the request to your Container Instance.

   You have successfully created and used Cognitive Services containers in Azure Container Instance.

What is Image Analysis?

6/22/2021 • 4 minutes to read • Edit Online

IMPORTANT

Transport Layer Security (TLS) 1.2 is now enforced for all HTTP requests to this service. For more information, see Azure Cognitive Services security.

The Computer Vision Image Analysis service can extract a wide variety of visual features from your images. For example, it can determine whether an image contains adult content, find specific brands or objects, or find human faces.

You can use Image Analysis through a client library SDK or by calling the REST API directly. Follow the quickstart to get started.

This documentation contains the following types of articles:

- The quickstarts are step-by-step instructions that let you make calls to the service and get results in a short period of time.
- The how-to guides contain instructions for using the service in more specific or customized ways.
- The conceptual articles provide in-depth explanations of the service's functionality and features.
- The tutorials are longer guides that show you how to use this service as a component in broader business solutions.

Image Analysis features

You can analyze images to provide insights about their visual features and characteristics. All of the features in the list below are provided by the Analyze Image API. Follow a quickstart to get started.

Tag visual features

Identify and tag visual features in an image, from a set of thousands of recognizable objects, living things, scenery, and actions. When the tags are ambiguous or not common knowledge, the API response provides hints to clarify the context of the tag. Tagging isn't limited to the main subject, such as a person in the foreground, but also includes the setting (indoor or outdoor), furniture, tools, plants, animals, accessories, gadgets, and so on. Tag visual features

Detect objects

Object detection is similar to tagging, but the API returns the bounding box coordinates for each tag applied. For example, if an image contains a dog, cat and person, the Detect operation will list those objects together with their coordinates in the image. You can use this functionality to process further relationships between the objects in an image. It also lets you know when there are multiple instances of the same tag in an image. Detect objects

Detect brands

Identify commercial brands in images or videos from a database of thousands of global logos. You can use this feature, for example, to discover which brands are most popular on social media or most prevalent in media product placement. Detect brands

Categorize an image

Identify and categorize an entire image, using a category taxonomy with parent/child hereditary hierarchies.

Categories can be used alone, or with our new tagging models.
Currently, English is the only supported language for tagging and categorizing images. Categorize an image

Describe an image

Generate a description of an entire image in human-readable language, using complete sentences. Computer Vision's algorithms generate various descriptions based on the objects identified in the image. The descriptions are each evaluated and a confidence score generated. A list is then returned ordered from highest confidence score to lowest. Describe an image

Detect faces

Detect faces in an image and provide information about each detected face. Computer Vision returns the coordinates, rectangle, gender, and age for each detected face.
Computer Vision provides a subset of the Face service functionality. You can use the Face service for more detailed analysis, such as facial identification and pose detection. Detect faces

Detect image types

Detect characteristics about an image, such as whether an image is a line drawing or the likelihood of whether an image is clip art. Detect image types

Detect domain-specific content

Use domain models to detect and identify domain-specific content in an image, such as celebrities and landmarks. For example, if an image contains people, Computer Vision can use a domain model for celebrities to determine if the people detected in the image are known celebrities. Detect domain-specific content

Detect the color scheme

Analyze color usage within an image. Computer Vision can determine whether an image is black & white or color and, for color images, identify the dominant and accent colors. Detect the color scheme

Generate a thumbnail

Analyze the contents of an image to generate an appropriate thumbnail for that image. Computer Vision first generates a high-quality thumbnail and then analyzes the objects within the image to determine the *area of interest*. Computer Vision then crops the image to fit the requirements of the area of interest. The generated thumbnail can be presented using an aspect ratio that is different from the aspect ratio of the original image, depending on your needs. Generate a thumbnail

Get the area of interest

Analyze the contents of an image to return the coordinates of the *area of interest*. Instead of cropping the image and generating a thumbnail, Computer Vision returns the bounding box coordinates of the region, so the calling application can modify the original image as desired. Get the area of interest

Moderate content in images

You can use Computer Vision to detect adult content in an image and return confidence scores for different classifications. The threshold for flagging content can be set on a sliding scale to accommodate your preferences.

Image requirements

Image Analysis works on images that meet the following requirements:

- The image must be presented in JPEG, PNG, GIF, or BMP format
- The file size of the image must be less than 4 megabytes (MB)
- The dimensions of the image must be greater than 50 x 50 pixels

Data privacy and security

As with all of the Cognitive Services, developers using the Computer Vision service should be aware of Microsoft's policies on customer data. See the Cognitive Services page on the Microsoft Trust Center to learn more.

Next steps

Get started with Image Analysis by following the quickstart guide in your preferred development language:

- Quickstart: Computer Vision REST API or client libraries

Quickstart: Use the Image Analysis client library or REST API

4/20/2021 • 51 minutes to read • Edit Online

Get started with the Image Analysis REST API or client libraries. The Analyze Image service provides you with AI algorithms for processing images and returning information on their visual features. Follow these steps to install a package to your application and try out the sample code for basic tasks.

Use the Image Analysis client library to analyze an image for tags, text description, faces, adult content, and more.

Reference documentation | Library source code | Package (NuGet) | Samples

Prerequisites

- An Azure subscription - Create one for free
- The Visual Studio IDE or current version of .NET Core.
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a new C# application

- Visual Studio IDE
- CLI

Using Visual Studio, create a new .NET Core application.

Install the client library

Once you've created a new project, install the client library by right-clicking on the project solution in the **Solution Explorer** and selecting **Manage NuGet Packages**. In the package manager that opens select **Browse**, check **Include prerelease**, and search for `Microsoft.Azure.CognitiveServices.Vision.ComputerVision`. Select version `7.0.0`, and then **Install**.

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

From the project directory, open the *Program.cs* file in your preferred editor or IDE. Add the following `using` directives:

```
using System;
using System.Collections.Generic;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
using System.Threading.Tasks;
using System.IO;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System.Threading;
using System.Linq;
```

In the application's **Program** class, create variables for your resource's Azure endpoint and key.

```
// Add your Computer Vision subscription key and endpoint
static string subscriptionKey = "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE";
static string endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE";
```

> **IMPORTANT**
>
> Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your key and endpoint in the resource's **key and endpoint** page, under **resource management**.
>
> Remember to remove the key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. See the Cognitive Services security article for more information.

In the application's `Main` method, add calls for the methods used in this quickstart. You will create these later.

```
// Create a client
ComputerVisionClient client = Authenticate(endpoint, subscriptionKey);

// Analyze an image to get features and other properties.
AnalyzeImageUrl(client, ANALYZE_URL_IMAGE).Wait();
```

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the Image Analysis .NET SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to do most image operations.
ComputerVisionClientExtensions	This class contains additional methods for the **ComputerVisionClient**.
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Image Analysis client library for .NET:

- Authenticate the client
- Analyze an image

Authenticate the client

> **NOTE**
> This quickstart assumes you've created environment variables for your Computer Vision key and endpoint, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT` respectively.

In a new method in the **Program** class, instantiate a client with your endpoint and key. Create a **ApiKeyServiceClientCredentials** object with your key, and use it with your endpoint to create a **ComputerVisionClient** object.

```
/*
 * AUTHENTICATE
 * Creates a Computer Vision client used by each example.
 */
public static ComputerVisionClient Authenticate(string endpoint, string key)
{
    ComputerVisionClient client =
      new ComputerVisionClient(new ApiKeyServiceClientCredentials(key))
      { Endpoint = endpoint };
    return client;
}
```

I authenticated the client I ran into an issue

Analyze an image

The following code defines a method, `AnalyzeImageUrl`, which uses the client object to analyze a remote image and print the results. The method returns a text description, categorization, list of tags, detected faces, adult content flags, main colors, and image type.

> **TIP**
> You can also analyze a local image. See the ComputerVisionClient methods, such as **AnalyzeImageInStreamAsync**. Or, see the sample code on GitHub for scenarios involving local images.

Set up test image

In your **Program** class, save a reference to the URL of the image you want to analyze.

```
// URL image used for analyzing an image (image of puppy)
private const string ANALYZE_URL_IMAGE =
"https://moderatorsampleimages.blob.core.windows.net/samples/sample16.png";
```

Specify visual features

Define your new method for image analysis. Add the code below, which specifies visual features you'd like to extract in your analysis. See the **VisualFeatureTypes** enum for a complete list.

```
/*
 * ANALYZE IMAGE - URL IMAGE
 * Analyze URL image. Extracts captions, categories, tags, objects, faces, racy/adult/gory content,
 * brands, celebrities, landmarks, color scheme, and image types.
 */
public static async Task AnalyzeImageUrl(ComputerVisionClient client, string imageUrl)
{
    Console.WriteLine("-----------------------------------------------------------");
    Console.WriteLine("ANALYZE IMAGE - URL");
    Console.WriteLine();

    // Creating a list that defines the features to be extracted from the image.

    List<VisualFeatureTypes?> features = new List<VisualFeatureTypes?>()
    {
        VisualFeatureTypes.Categories, VisualFeatureTypes.Description,
        VisualFeatureTypes.Faces, VisualFeatureTypes.ImageType,
        VisualFeatureTypes.Tags, VisualFeatureTypes.Adult,
        VisualFeatureTypes.Color, VisualFeatureTypes.Brands,
        VisualFeatureTypes.Objects
    };
```

Call the Analyze API

The **AnalyzeImageAsync** method returns an **ImageAnalysis** object that contains all of extracted information.

```
Console.WriteLine($"Analyzing the image {Path.GetFileName(imageUrl)}...");
Console.WriteLine();
// Analyze the URL image
ImageAnalysis results = await client.AnalyzeImageAsync(imageUrl, visualFeatures: features);
```

The following sections show how to parse this information in detail.

Insert any of the following code blocks into your **AnalyzeImageUrl** method to parse data from the visual features you requested above. Remember to add a closing bracket at the end.

```
}
```

Get image description

The following code gets the list of generated captions for the image. See Describe images for more details.

```
// Sunmarizes the image content.
Console.WriteLine("Summary:");
foreach (var caption in results.Description.Captions)
{
    Console.WriteLine($"{caption.Text} with confidence {caption.Confidence}");
}
Console.WriteLine();
```

Get image category

The following code gets the detected category of the image. See Categorize images for more details.

```
// Display categories the image is divided into.
Console.WriteLine("Categories:");
foreach (var category in results.Categories)
{
    Console.WriteLine($"{category.Name} with confidence {category.Score}");
}
Console.WriteLine();
```

Get image tags

The following code gets the set of detected tags in the image. See Content tags for more details.

```
// Image tags and their confidence score
Console.WriteLine("Tags:");
foreach (var tag in results.Tags)
{
    Console.WriteLine($"{tag.Name} {tag.Confidence}");
}
Console.WriteLine();
```

Detect objects

The following code detects common objects in the image and prints them to the console. See Object detection for more details.

```
// Objects
Console.WriteLine("Objects:");
foreach (var obj in results.Objects)
{
    Console.WriteLine($"{obj.ObjectProperty} with confidence {obj.Confidence} at location {obj.Rectangle.X},
" +
      $"{obj.Rectangle.X + obj.Rectangle.W}, {obj.Rectangle.Y}, {obj.Rectangle.Y + obj.Rectangle.H}");
}
Console.WriteLine();
```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. See Brand detection for more details.

```
// Well-known (or custom, if set) brands.
Console.WriteLine("Brands:");
foreach (var brand in results.Brands)
{
    Console.WriteLine($"Logo of {brand.Name} with confidence {brand.Confidence} at location
{brand.Rectangle.X}, " +
      $"{brand.Rectangle.X + brand.Rectangle.W}, {brand.Rectangle.Y}, {brand.Rectangle.Y +
brand.Rectangle.H}");
}
Console.WriteLine();
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and select face attributes. See Face detection for more details.

```
// Faces
Console.WriteLine("Faces:");
foreach (var face in results.Faces)
{
    Console.WriteLine($"A {face.Gender} of age {face.Age} at location {face.FaceRectangle.Left}, " +
      $"{face.FaceRectangle.Left}, {face.FaceRectangle.Top + face.FaceRectangle.Width}, " +
      $"{face.FaceRectangle.Top + face.FaceRectangle.Height}");
}
Console.WriteLine();
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. See Adult, racy, gory content for more details.

```
// Adult or racy content, if any.
Console.WriteLine("Adult:");
Console.WriteLine($"Has adult content: {results.Adult.IsAdultContent} with confidence
{results.Adult.AdultScore}");
Console.WriteLine($"Has racy content: {results.Adult.IsRacyContent} with confidence
{results.Adult.RacyScore}");
Console.WriteLine($"Has gory content: {results.Adult.IsGoryContent} with confidence
{results.Adult.GoreScore}");
Console.WriteLine();
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. See Color schemes for more details.

```
// Identifies the color scheme.
Console.WriteLine("Color Scheme:");
Console.WriteLine("Is black and white?: " + results.Color.IsBWImg);
Console.WriteLine("Accent color: " + results.Color.AccentColor);
Console.WriteLine("Dominant background color: " + results.Color.DominantColorBackground);
Console.WriteLine("Dominant foreground color: " + results.Color.DominantColorForeground);
Console.WriteLine("Dominant colors: " + string.Join(",", results.Color.DominantColors));
Console.WriteLine();
```

Get domain-specific content

Image Analysis can use specialized models to do further analysis on images. See Domain-specific content for more details.

The following code parses data about detected celebrities in the image.

```
// Celebrities in image, if any.
Console.WriteLine("Celebrities:");
foreach (var category in results.Categories)
{
    if (category.Detail?.Celebrities != null)
    {
        foreach (var celeb in category.Detail.Celebrities)
        {
            Console.WriteLine($"{celeb.Name} with confidence {celeb.Confidence} at location
{celeb.FaceRectangle.Left}, " +
              $"{celeb.FaceRectangle.Top}, {celeb.FaceRectangle.Height}, {celeb.FaceRectangle.Width}");
        }
    }
}
Console.WriteLine();
```

The following code parses data about detected landmarks in the image.

```
// Popular landmarks in image, if any.
Console.WriteLine("Landmarks:");
foreach (var category in results.Categories)
{
    if (category.Detail?.Landmarks != null)
    {
        foreach (var landmark in category.Detail.Landmarks)
        {
            Console.WriteLine($"{landmark.Name} with confidence {landmark.Confidence}");
        }
    }
}
Console.WriteLine();
```

Get the image type

The following code prints information about the type of image—whether it is clip art or a line drawing.

```
// Detects the image types.
Console.WriteLine("Image Type:");
Console.WriteLine("Clip Art Type: " + results.ImageType.ClipArtType);
Console.WriteLine("Line Drawing Type: " + results.ImageType.LineDrawingType);
Console.WriteLine();
```

I analyzed an image I ran into an issue

Run the application

- Visual Studio IDE
- CLI

Run the application by clicking the **Debug** button at the top of the IDE window.

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

Next steps

In this quickstart, you learned how to install the Image Analysis client library and make basic image analysis calls. Next, learn more about the Analyze API features.

Call the Analyze API

- Image Analysis overview
- The source code for this sample can be found on GitHub.

Use the Image Analysis client library to analyze an image for tags, text description, faces, adult content, and more.

Reference documentation | Library source code | Package (PiPy) | Samples

Prerequisites

- An Azure subscription - Create one for free
- Python 3.x
  - Your Python installation should include pip. You can check if you have pip installed by running `pip --version` on the command line. Get pip by installing the latest version of Python.
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Install the client library

You can install the client library with:

```
pip install --upgrade azure-cognitiveservices-vision-computervision
```

Also install the Pillow library.

```
pip install pillow
```

Create a new Python application

Create a new Python file—*quickstart-file.py*, for example. Then open it in your preferred editor or IDE and import the following libraries.

```
from azure.cognitiveservices.vision.computervision import ComputerVisionClient
from azure.cognitiveservices.vision.computervision.models import OperationStatusCodes
from azure.cognitiveservices.vision.computervision.models import VisualFeatureTypes
from msrest.authentication import CognitiveServicesCredentials

from array import array
import os
from PIL import Image
import sys
import time
```

> **TIP**
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Then, create variables for your resource's Azure endpoint and key.

```
subscription_key = "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE"
endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE"
```

> **IMPORTANT**
>
> Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your key and endpoint in the resource's **key and endpoint** page, under **resource management**.
>
> Remember to remove the key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. For example, Azure key vault.

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the Image Analysis Python SDK.

NAME	DESCRIPTION
ComputerVisionClientOperationsMixin	This class directly handles all of the image operations, such as image analysis, text detection, and thumbnail generation.
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes. It implements **ComputerVisionClientOperationsMixin**.
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of **VisualFeatureTypes** values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Image Analysis client library for Python:

- Authenticate the client
- Analyze an image

Authenticate the client

Instantiate a client with your endpoint and key. Create a CognitiveServicesCredentials object with your key, and use it with your endpoint to create a ComputerVisionClient object.

```
computervision_client = ComputerVisionClient(endpoint, CognitiveServicesCredentials(subscription_key))
```

I authenticated the client I ran into an issue

Analyze an image

Use your client object to analyze the visual features of a remote image. First save a reference to the URL of an image you want to analyze.

```
remote_image_url = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/landmark.jpg"
```

TIP

You can also analyze a local image. See the ComputerVisionClientOperationsMixin methods, such as **analyze_image_in_stream**. Or, see the sample code on GitHub for scenarios involving local images.

Get image description

The following code gets the list of generated captions for the image. See Describe images for more details.

```
'''
Describe an Image - remote
This example describes the contents of an image with the confidence score.
'''
print("===== Describe an image - remote =====")
# Call API
description_results = computervision_client.describe_image(remote_image_url )

# Get the captions (descriptions) from the response, with confidence level
print("Description of remote image: ")
if (len(description_results.captions) == 0):
    print("No description detected.")
else:
    for caption in description_results.captions:
        print("'{}' with confidence {:.2f}%".format(caption.text, caption.confidence * 100))
```

Get image category

The following code gets the detected category of the image. See Categorize images for more details.

```
'''
Categorize an Image - remote
This example extracts (general) categories from a remote image with a confidence score.
'''
print("===== Categorize an image - remote =====")
# Select the visual feature(s) you want.
remote_image_features = ["categories"]
# Call API with URL and features
categorize_results_remote = computervision_client.analyze_image(remote_image_url , remote_image_features)

# Print results with confidence score
print("Categories from remote image: ")
if (len(categorize_results_remote.categories) == 0):
    print("No categories detected.")
else:
    for category in categorize_results_remote.categories:
        print("'{}' with confidence {:.2f}%".format(category.name, category.score * 100))
```

Get image tags

The following code gets the set of detected tags in the image. See Content tags for more details.

```
'''
Tag an Image - remote
This example returns a tag (key word) for each thing in the image.
'''
print("===== Tag an image - remote =====")
# Call API with remote image
tags_result_remote = computervision_client.tag_image(remote_image_url )

# Print results with confidence score
print("Tags in the remote image: ")
if (len(tags_result_remote.tags) == 0):
    print("No tags detected.")
else:
    for tag in tags_result_remote.tags:
        print("'{}' with confidence {:.2f}%".format(tag.name, tag.confidence * 100))
```

Detect objects

The following code detects common objects in the image and prints them to the console. See Object detection for more details.

```
'''
Detect Objects - remote
This example detects different kinds of objects with bounding boxes in a remote image.
'''
print("===== Detect Objects - remote =====")
# Get URL image with different objects
remote_image_url_objects = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/objects.jpg"
# Call API with URL
detect_objects_results_remote = computervision_client.detect_objects(remote_image_url_objects)

# Print detected objects results with bounding boxes
print("Detecting objects in remote image:")
if len(detect_objects_results_remote.objects) == 0:
    print("No objects detected.")
else:
    for object in detect_objects_results_remote.objects:
        print("object at location {}, {}, {}, {}".format( \
        object.rectangle.x, object.rectangle.x + object.rectangle.w, \
        object.rectangle.y, object.rectangle.y + object.rectangle.h))
```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. See Brand detection for more details.

```
'''
Detect Brands - remote
This example detects common brands like logos and puts a bounding box around them.
'''
print("===== Detect Brands - remote =====")
# Get a URL with a brand logo
remote_image_url = "https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/images/gray-shirt-logo.jpg"
# Select the visual feature(s) you want
remote_image_features = ["brands"]
# Call API with URL and features
detect_brands_results_remote = computervision_client.analyze_image(remote_image_url, remote_image_features)

print("Detecting brands in remote image: ")
if len(detect_brands_results_remote.brands) == 0:
    print("No brands detected.")
else:
    for brand in detect_brands_results_remote.brands:
        print("'{}' brand detected with confidence {:.1f}% at location {}, {}, {}, {}".format( \
        brand.name, brand.confidence * 100, brand.rectangle.x, brand.rectangle.x + brand.rectangle.w, \
        brand.rectangle.y, brand.rectangle.y + brand.rectangle.h))
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and select face attributes. See Face detection for more details.

```
'''
Detect Faces - remote
This example detects faces in a remote image, gets their gender and age,
and marks them with a bounding box.
'''
print("===== Detect Faces - remote =====")
# Get an image with faces
remote_image_url_faces = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/faces.jpg"
# Select the visual feature(s) you want.
remote_image_features = ["faces"]
# Call the API with remote URL and features
detect_faces_results_remote = computervision_client.analyze_image(remote_image_url_faces,
remote_image_features)

# Print the results with gender, age, and bounding box
print("Faces in the remote image: ")
if (len(detect_faces_results_remote.faces) == 0):
    print("No faces detected.")
else:
    for face in detect_faces_results_remote.faces:
        print("'{}' of age {} at location {}, {}, {}, {}".format(face.gender, face.age, \
        face.face_rectangle.left, face.face_rectangle.top, \
        face.face_rectangle.left + face.face_rectangle.width, \
        face.face_rectangle.top + face.face_rectangle.height))
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. See Adult, racy, gory content for more details.

```
'''
Detect Adult or Racy Content - remote
This example detects adult or racy content in a remote image, then prints the adult/racy score.
The score is ranged 0.0 - 1.0 with smaller numbers indicating negative results.
'''
print("===== Detect Adult or Racy Content - remote =====")
# Select the visual feature(s) you want
remote_image_features = ["adult"]
# Call API with URL and features
detect_adult_results_remote = computervision_client.analyze_image(remote_image_url, remote_image_features)

# Print results with adult/racy score
print("Analyzing remote image for adult or racy content ... ")
print("Is adult content: {} with confidence
{:.2f}".format(detect_adult_results_remote.adult.is_adult_content,
detect_adult_results_remote.adult.adult_score * 100))
print("Has racy content: {} with confidence
{:.2f}".format(detect_adult_results_remote.adult.is_racy_content,
detect_adult_results_remote.adult.racy_score * 100))
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. See Color schemes for more details.

```
'''
Detect Color - remote
This example detects the different aspects of its color scheme in a remote image.
'''
print("===== Detect Color - remote =====")
# Select the feature(s) you want
remote_image_features = ["color"]
# Call API with URL and features
detect_color_results_remote = computervision_client.analyze_image(remote_image_url, remote_image_features)

# Print results of color scheme
print("Getting color scheme of the remote image: ")
print("Is black and white: {}".format(detect_color_results_remote.color.is_bw_img))
print("Accent color: {}".format(detect_color_results_remote.color.accent_color))
print("Dominant background color: {}".format(detect_color_results_remote.color.dominant_color_background))
print("Dominant foreground color: {}".format(detect_color_results_remote.color.dominant_color_foreground))
print("Dominant colors: {}".format(detect_color_results_remote.color.dominant_colors))
```

Get domain-specific content

Image Analysis can use specialized model to do further analysis on images. See Domain-specific content for more details.

The following code parses data about detected celebrities in the image.

```
'''
Detect Domain-specific Content - remote
This example detects celebrites and landmarks in remote images.
'''
print("===== Detect Domain-specific Content - remote =====")
# URL of one or more celebrities
remote_image_url_celebs = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-
files/master/ComputerVision/Images/faces.jpg"
# Call API with content type (celebrities) and URL
detect_domain_results_celebs_remote = computervision_client.analyze_image_by_domain("celebrities",
remote_image_url_celebs)

# Print detection results with name
print("Celebrities in the remote image:")
if len(detect_domain_results_celebs_remote.result["celebrities"]) == 0:
    print("No celebrities detected.")
else:
    for celeb in detect_domain_results_celebs_remote.result["celebrities"]:
        print(celeb["name"])
```

The following code parses data about detected landmarks in the image.

```
# Call API with content type (landmarks) and URL
detect_domain_results_landmarks = computervision_client.analyze_image_by_domain("landmarks",
remote_image_url)
print()

print("Landmarks in the remote image:")
if len(detect_domain_results_landmarks.result["landmarks"]) == 0:
    print("No landmarks detected.")
else:
    for landmark in detect_domain_results_landmarks.result["landmarks"]:
        print(landmark["name"])
```

Get the image type

The following code prints information about the type of image—whether it is clip art or line drawing.

```
'''
Detect Image Types - remote
This example detects an image's type (clip art/line drawing).
'''
print("===== Detect Image Types - remote =====")
# Get URL of an image with a type
remote_image_url_type = "https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-
files/master/ComputerVision/Images/type-image.jpg"
# Select visual feature(s) you want
remote_image_features = [VisualFeatureTypes.image_type]
# Call API with URL and features
detect_type_results_remote = computervision_client.analyze_image(remote_image_url_type,
remote_image_features)

# Prints type results with degree of accuracy
print("Type of remote image:")
if detect_type_results_remote.image_type.clip_art_type == 0:
    print("Image is not clip art.")
elif detect_type_results_remote.image_type.line_drawing_type == 1:
    print("Image is ambiguously clip art.")
elif detect_type_results_remote.image_type.line_drawing_type == 2:
    print("Image is normal clip art.")
else:
    print("Image is good clip art.")

if detect_type_results_remote.image_type.line_drawing_type == 0:
    print("Image is not a line drawing.")
else:
    print("Image is a line drawing")
```

I analyzed an image I ran into an issue

Run the application

Run the application with the `python` command on your quickstart file.

```
python quickstart-file.py
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the Image Analysis client library and make basic image analysis calls. Next, learn more about the Analyze API features.

Call the Analyze API

- Image Analysis overview
- The source code for this sample can be found on GitHub.

Use the Image Analysis client library to analyze an image for tags, text description, faces, adult content, and more.

Reference documentation | Library source code |Artifact (Maven) | Samples

Prerequisites

- An Azure subscription - Create one for free
- The current version of the Java Development Kit (JDK)
- The Gradle build tool, or another dependency manager.
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a new Gradle project

In a console window (such as cmd, PowerShell, or Bash), create a new directory for your app, and navigate to it.

```
mkdir myapp && cd myapp
```

Run the `gradle init` command from your working directory. This command will create essential build files for Gradle, including *build.gradle.kts*, which is used at runtime to create and configure your application.

```
gradle init --type basic
```

When prompted to choose a **DSL**, select **Kotlin**.

Install the client library

This quickstart uses the Gradle dependency manager. You can find the client library and information for other dependency managers on the Maven Central Repository.

Locate *build.gradle.kts* and open it with your preferred IDE or text editor. Then copy in the following build configuration. This configuration defines the project as a Java application whose entry point is the class **ImageAnalysisQuickstart**. It imports the Computer Vision library.

```
plugins {
    java
    application
}
application {
    mainClass.set("ImageAnalysisQuickstart")
}
repositories {
    mavenCentral()
}
dependencies {
    implementation(group = "com.microsoft.azure.cognitiveservices", name = "azure-cognitiveservices-
computervision", version = "1.0.6-beta")
}
```

Create a Java file

From your working directory, run the following command to create a project source folder:

```
mkdir -p src/main/java
```

Navigate to the new folder and create a file called *ImageAnalysisQuickstart.java*. Open it in your preferred editor or IDE and add the following `import` statements:

```
import com.microsoft.azure.cognitiveservices.vision.computervision.*;
import com.microsoft.azure.cognitiveservices.vision.computervision.implementation.ComputerVisionImpl;
import com.microsoft.azure.cognitiveservices.vision.computervision.models.*;

import java.io.*;
import java.nio.file.Files;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
```

> **TIP**
>
> Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Define the class **ImageAnalysisQuickstart**.

```
public class ComputerVisionQuickstart {
```

```
}
```

Within the **ImageAnalysisQuickstart** class, create variables for your resource's key and endpoint.

```
static String subscriptionKey = "PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE";
static String endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE";
```

> **IMPORTANT**
>
> Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your key and endpoint in the resource's **key and endpoint** page, under **resource management**.
>
> Remember to remove the key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. See the Cognitive Services security article for more information.

In the application's **main** method, add calls for the methods used in this quickstart. You'll define these later.

```
public static void main(String[] args) {

    System.out.println("\nAzure Cognitive Services Computer Vision - Java Quickstart Sample");

    // Create an authenticated Computer Vision client.
    ComputerVisionClient compVisClient = Authenticate(subscriptionKey, endpoint);

    // Analyze local and remote images
    AnalyzeLocalImage(compVisClient);

}
```

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the Image Analysis Java SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes.
ComputerVision	This class comes from the client object and directly handles all of the image operations, such as image analysis, text detection, and thumbnail generation.
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Image Analysis client library for Java:

- Authenticate the client
- Analyze an image

Authenticate the client

In a new method, instantiate a ComputerVisionClient object with your endpoint and key.

```
public static ComputerVisionClient Authenticate(String subscriptionKey, String endpoint){
    return ComputerVisionManager.authenticate(subscriptionKey).withEndpoint(endpoint);
}
```

I authenticated the client I ran into an issue

Analyze an image

The following code defines a method, `AnalyzeLocalImage`, which uses the client object to analyze a local image and print the results. The method returns a text description, categorization, list of tags, detected faces, adult content flags, main colors, and image type.

TIP

You can also analyze a remote image using its URL. See the ComputerVision methods, such as **AnalyzeImage**. Or, see the sample code on GitHub for scenarios involving remote images.

Set up test image

First, create a **resources/** folder in the **src/main/** folder of your project, and add an image you'd like to analyze. Then add the following method definition to your **ImageAnalysisQuickstart** class. Change the value of the `pathToLocalImage` to match your image file.

```
public static void AnalyzeLocalImage(ComputerVisionClient compVisClient) {
    /*
     * Analyze a local image:
     *
     * Set a string variable equal to the path of a local image. The image path
     * below is a relative path.
     */
    String pathToLocalImage = "src\\main\\resources\\myImage.png";
```

Specify visual features

Next, specify which visual features you'd like to extract in your analysis. See the VisualFeatureTypes enum for a complete list.

```
// This list defines the features to be extracted from the image.
List<VisualFeatureTypes> featuresToExtractFromLocalImage = new ArrayList<>();
featuresToExtractFromLocalImage.add(VisualFeatureTypes.DESCRIPTION);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.CATEGORIES);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.TAGS);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.FACES);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.OBJECTS);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.BRANDS);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.ADULT);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.COLOR);
featuresToExtractFromLocalImage.add(VisualFeatureTypes.IMAGE_TYPE);
```

Analyze

This block prints detailed results to the console for each scope of image analysis. The **analyzeImageInStream** method returns an **ImageAnalysis** object that contains all of extracted information.

```
try {
    // Need a byte array for analyzing a local image.
    File rawImage = new File(pathToLocalImage);
    byte[] imageByteArray = Files.readAllBytes(rawImage.toPath());

    // Call the Computer Vision service and tell it to analyze the loaded image.
    ImageAnalysis analysis = compVisClient.computerVision().analyzeImageInStream().withImage(imageByteArray)
            .withVisualFeatures(featuresToExtractFromLocalImage).execute();
```

The following sections show how to parse this information in detail.

Get image description

The following code gets the list of generated captions for the image. For more information, see Describe images.

```
// Display image captions and confidence values.
System.out.println("\nCaptions: ");
for (ImageCaption caption : analysis.description().captions()) {
    System.out.printf("\'%s\' with confidence %f\n", caption.text(), caption.confidence());
}
```

Get image category

The following code gets the detected category of the image. For more information, see Categorize images.

```
// Display image category names and confidence values.
System.out.println("\nCategories: ");
for (Category category : analysis.categories()) {
    System.out.printf("\'%s\' with confidence %f\n", category.name(), category.score());
}
```

Get image tags

The following code gets the set of detected tags in the image. For more information, see Content tags.

```
// Display image tags and confidence values.
System.out.println("\nTags: ");
for (ImageTag tag : analysis.tags()) {
    System.out.printf("\'%s\' with confidence %f\n", tag.name(), tag.confidence());
}
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and selects face attributes. For more information, see Face detection.

```
// Display any faces found in the image and their location.
System.out.println("\nFaces: ");
for (FaceDescription face : analysis.faces()) {
    System.out.printf("\'%s\' of age %d at location (%d, %d), (%d, %d)\n", face.gender(), face.age(),
            face.faceRectangle().left(), face.faceRectangle().top(),
            face.faceRectangle().left() + face.faceRectangle().width(),
            face.faceRectangle().top() + face.faceRectangle().height());
}
```

Detect objects

The following code returns the detected objects in the image with their coordinates. For more information, see Object detection.

```
// Display any objects found in the image.
System.out.println("\nObjects: ");
for ( DetectedObject object : analysis.objects()) {
    System.out.printf("Object \'%s\' detected at location (%d, %d)\n", object.objectProperty(),
            object.rectangle().x(), object.rectangle().y());
}
```

Detect brands

The following code returns the detected brand logos in the image with their coordinates. For more information, see Brand detection.

```
// Display any brands found in the image.
System.out.println("\nBrands: ");
for ( DetectedBrand brand : analysis.brands()) {
    System.out.printf("Brand \'%s\' detected at location (%d, %d)\n", brand.name(),
            brand.rectangle().x(), brand.rectangle().y());
}
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. For more information, see Adult, racy, gory content.

```
// Display whether any adult/racy/gory content was detected and the confidence
// values.
System.out.println("\nAdult: ");
System.out.printf("Is adult content: %b with confidence %f\n", analysis.adult().isAdultContent(),
        analysis.adult().adultScore());
System.out.printf("Has racy content: %b with confidence %f\n", analysis.adult().isRacyContent(),
        analysis.adult().racyScore());
System.out.printf("Has gory content: %b with confidence %f\n", analysis.adult().isGoryContent(),
        analysis.adult().goreScore());
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. For more information, see Color schemes.

```
// Display the image color scheme.
System.out.println("\nColor scheme: ");
System.out.println("Is black and white: " + analysis.color().isBWImg());
System.out.println("Accent color: " + analysis.color().accentColor());
System.out.println("Dominant background color: " + analysis.color().dominantColorBackground());
System.out.println("Dominant foreground color: " + analysis.color().dominantColorForeground());
System.out.println("Dominant colors: " + String.join(", ", analysis.color().dominantColors()));
```

Get domain-specific content

Image Analysis can use specialized model to do further analysis on images. For more information, see Domain-specific content.

The following code parses data about detected celebrities in the image.

```
// Display any celebrities detected in the image and their locations.
System.out.println("\nCelebrities: ");
for (Category category : analysis.categories()) {
    if (category.detail() != null && category.detail().celebrities() != null) {
        for (CelebritiesModel celeb : category.detail().celebrities()) {
            System.out.printf("\'%s\' with confidence %f at location (%d, %d), (%d, %d)\n", celeb.name(),
                    celeb.confidence(), celeb.faceRectangle().left(), celeb.faceRectangle().top(),
                    celeb.faceRectangle().left() + celeb.faceRectangle().width(),
                    celeb.faceRectangle().top() + celeb.faceRectangle().height());
        }
    }
}
```

The following code parses data about detected landmarks in the image.

```
// Display any landmarks detected in the image and their locations.
System.out.println("\nLandmarks: ");
for (Category category : analysis.categories()) {
    if (category.detail() != null && category.detail().landmarks() != null) {
        for (LandmarksModel landmark : category.detail().landmarks()) {
            System.out.printf("\'%s\' with confidence %f\n", landmark.name(), landmark.confidence());
        }
    }
}
```

Get the image type

The following code prints information about the type of image—whether it is clip art or line drawing.

```
// Display what type of clip art or line drawing the image is.
System.out.println("\nImage type:");
System.out.println("Clip art type: " + analysis.imageType().clipArtType());
System.out.println("Line drawing type: " + analysis.imageType().lineDrawingType());
```

I analyzed an image I ran into an issue

Close out the method

Complete the try/catch block and close the method.

```
    }

    catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

Run the application

You can build the app with:

```
gradle build
```

Run the application with the `gradle run` command:

```
gradle run
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the Image Analysis client library and make basic image analysis calls. Next, learn more about the Analyze API features.

Call the Analyze API

- Image Analysis overview
- The source code for this sample can be found on GitHub.

Use the Image Analysis client library to analyze an image for tags, text description, faces, adult content, and more.

Reference documentation | Library source code | Package (npm) | Samples

Prerequisites

- An Azure subscription - Create one for free
- The current version of Node.js
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a new Node.js application

In a console window (such as cmd, PowerShell, or Bash), create a new directory for your app, and navigate to it.

```
mkdir myapp && cd myapp
```

Run the `npm init` command to create a node application with a `package.json` file.

```
npm init
```

Install the client library

Install the `ms-rest-azure` and `@azure/cognitiveservices-computervision` NPM package:

```
npm install @azure/cognitiveservices-computervision
```

Also install the async module:

```
npm install async
```

Your app's `package.json` file will be updated with the dependencies.

Create a new file, *index.js*, and open it in a text editor. Add the following import statements.

```
'use strict';

const async = require('async');
const fs = require('fs');
const https = require('https');
const path = require("path");
const createReadStream = require('fs').createReadStream
const sleep = require('util').promisify(setTimeout);
const ComputerVisionClient = require('@azure/cognitiveservices-computervision').ComputerVisionClient;
const ApiKeyCredentials = require('@azure/ms-rest-js').ApiKeyCredentials;
```

TIP

Want to view the whole quickstart code file at once? You can find it on GitHub, which contains the code examples in this quickstart.

Create variables for your resource's Azure endpoint and key.

```
/**
 * AUTHENTICATE
 * This single client is used for all examples.
 */
const key = 'PASTE_YOUR_COMPUTER_VISION_SUBSCRIPTION_KEY_HERE';
const endpoint = 'PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE';
```

IMPORTANT

Go to the Azure portal. If the Computer Vision resource you created in the **Prerequisites** section deployed successfully, click the **Go to Resource** button under **Next Steps**. You can find your key and endpoint in the resource's **key and endpoint** page, under **resource management**.

Remember to remove the key from your code when you're done, and never post it publicly. For production, consider using a secure way of storing and accessing your credentials. See the Cognitive Services security article for more information.

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the Image Analysis Node.js SDK.

NAME	DESCRIPTION
ComputerVisionClient	This class is needed for all Computer Vision functionality. You instantiate it with your subscription information, and you use it to do most image operations.
VisualFeatureTypes	This enum defines the different types of image analysis that can be done in a standard Analyze operation. You specify a set of **VisualFeatureTypes** values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Image Analysis client library for Node.js:

- Authenticate the client
- Analyze an image

Authenticate the client

Instantiate a client with your endpoint and key. Create a ApiKeyCredentials object with your key and endpoint, and use it to create a ComputerVisionClient object.

```
const computerVisionClient = new ComputerVisionClient(
  new ApiKeyCredentials({ inHeader: { 'Ocp-Apim-Subscription-Key': key } }), endpoint);
```

Then, define a function `computerVision` and declare an async series with primary function and callback function. You will add your quickstart code into the primary function, and call `computerVision` at the bottom of the script. The rest of the code in this quickstart goes inside the `computerVision` function.

```
function computerVision() {
  async.series([
    async function () {
```

```
    },
    function () {
      return new Promise((resolve) => {
        resolve();
      })
    }
  ], (err) => {
    throw (err);
  });
}

computerVision();
```

I authenticated the client I ran into an issue

Analyze an image

The code in this section analyzes remote images to extract various visual features. You can do these operations as part of the **analyzeImage** method of the client object, or you can call them using individual methods. See the reference documentation for details.

> **NOTE**
>
> You can also analyze a local image. See the ComputerVisionClient methods, such as **describeImageInStream**. Or, see the sample code on GitHub for scenarios involving local images.

Get image description

The following code gets the list of generated captions for the image. See Describe images for more details.

First, define the URL of an image to analyze:

```
const describeURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/celebrities.jpg';
```

Then add the following code to get the image description and print it to the console.

```
// Analyze URL image
console.log('Analyzing URL image to describe...', describeURL.split('/').pop());
const caption = (await computerVisionClient.describeImage(describeURL)).captions[0];
console.log(`This may be ${caption.text} (${caption.confidence.toFixed(2)} confidence)`);
```

Get image category

The following code gets the detected category of the image. See Categorize images for more details.

```
const categoryURLImage = 'https://moderatorsampleimages.blob.core.windows.net/samples/sample16.png';

// Analyze URL image
console.log('Analyzing category in image...', categoryURLImage.split('/').pop());
const categories = (await computerVisionClient.analyzeImage(categoryURLImage)).categories;
console.log(`Categories: ${formatCategories(categories)}`);
```

Define the helper function `formatCategories`:

```
// Formats the image categories
function formatCategories(categories) {
  categories.sort((a, b) => b.score - a.score);
  return categories.map(cat => `${cat.name} (${cat.score.toFixed(2)})`).join(', ');
}
```

Get image tags

The following code gets the set of detected tags in the image. See Content tags for more details.

```
console.log('-------------------------------------------------');
console.log('DETECT TAGS');
console.log();

// Image of different kind of dog.
const tagsURL = 'https://moderatorsampleimages.blob.core.windows.net/samples/sample16.png';

// Analyze URL image
console.log('Analyzing tags in image...', tagsURL.split('/').pop());
const tags = (await computerVisionClient.analyzeImage(tagsURL, { visualFeatures: ['Tags'] })).tags;
console.log(`Tags: ${formatTags(tags)}`);
```

Define the helper function `formatTags`:

```
// Format tags for display
function formatTags(tags) {
  return tags.map(tag => (`${tag.name} (${tag.confidence.toFixed(2)})`)).join(', ');
}
```

Detect objects

The following code detects common objects in the image and prints them to the console. See Object detection for more details.

```
// Image of a dog
const objectURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-node-sdk-
samples/master/Data/image.jpg';

// Analyze a URL image
console.log('Analyzing objects in image...', objectURL.split('/').pop());
const objects = (await computerVisionClient.analyzeImage(objectURL, { visualFeatures: ['Objects']
})).objects;
console.log();

// Print objects bounding box and confidence
if (objects.length) {
  console.log(`${objects.length} object${objects.length == 1 ? '' : 's'} found:`);
  for (const obj of objects) { console.log(`    ${obj.object} (${obj.confidence.toFixed(2)}) at
${formatRectObjects(obj.rectangle)}`); }
} else { console.log('No objects found.'); }
```

Define the helper function `formatRectObjects` to return the top, left, bottom, and right coordinates, along with the width and height.

```
// Formats the bounding box
function formatRectObjects(rect) {
  return `top=${rect.y}`.padEnd(10) + `left=${rect.x}`.padEnd(10) + `bottom=${rect.y + rect.h}`.padEnd(12)
    + `right=${rect.x + rect.w}`.padEnd(10) + `(${rect.w}x${rect.h})`;
}
```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. See Brand detection for more details.

```
const brandURLImage = 'https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/images/red-
shirt-logo.jpg';

// Analyze URL image
console.log('Analyzing brands in image...', brandURLImage.split('/').pop());
const brands = (await computerVisionClient.analyzeImage(brandURLImage, { visualFeatures: ['Brands']
})).brands;

// Print the brands found
if (brands.length) {
  console.log(`${brands.length} brand${brands.length != 1 ? 's' : ''} found:`);
  for (const brand of brands) {
    console.log(`    ${brand.name} (${brand.confidence.toFixed(2)} confidence)`);
  }
} else { console.log(`No brands found.`); }
```

Detect faces

The following code returns the detected faces in the image with their rectangle coordinates and select face attributes. See Face detection for more details.

```
const facesImageURL = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/faces.jpg';

// Analyze URL image.
console.log('Analyzing faces in image...', facesImageURL.split('/').pop());
// Get the visual feature for 'Faces' only.
const faces = (await computerVisionClient.analyzeImage(facesImageURL, { visualFeatures: ['Faces'] })).faces;

// Print the bounding box, gender, and age from the faces.
if (faces.length) {
  console.log(`${faces.length} face${faces.length == 1 ? '' : 's'} found:`);
  for (const face of faces) {
    console.log(`    Gender: ${face.gender}`.padEnd(20)
      + ` Age: ${face.age}`.padEnd(10) + `at ${formatRectFaces(face.faceRectangle)}`);
  }
} else { console.log('No faces found.'); }
```

Define the helper function `formatRectFaces`:

```
// Formats the bounding box
function formatRectFaces(rect) {
  return `top=${rect.top}`.padEnd(10) + `left=${rect.left}`.padEnd(10) + `bottom=${rect.top + rect.height}`.padEnd(12)
    + `right=${rect.left + rect.width}`.padEnd(10) + `(${rect.width}x${rect.height})`;
}
```

Detect adult, racy, or gory content

The following code prints the detected presence of adult content in the image. See Adult, racy, gory content for more details.

Define the URL of the image to use:

```
// The URL image and local images are not racy/adult.
// Try your own racy/adult images for a more effective result.
const adultURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/celebrities.jpg';
```

Then add the following code to detect adult content and print the results to the console.

```
// Function to confirm racy or not
const isIt = flag => flag ? 'is' : "isn't";

// Analyze URL image
console.log('Analyzing image for racy/adult content...', adultURLImage.split('/').pop());
const adult = (await computerVisionClient.analyzeImage(adultURLImage, {
  visualFeatures: ['Adult']
})).adult;
console.log(`This probably ${isIt(adult.isAdultContent)} adult content (${adult.adultScore.toFixed(4)} score)`);
console.log(`This probably ${isIt(adult.isRacyContent)} racy content (${adult.racyScore.toFixed(4)} score)`);
```

Get image color scheme

The following code prints the detected color attributes in the image, like the dominant colors and accent color. See Color schemes for more details.

```
const colorURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/celebrities.jpg';

// Analyze URL image
console.log('Analyzing image for color scheme...', colorURLImage.split('/').pop());
console.log();
const color = (await computerVisionClient.analyzeImage(colorURLImage, { visualFeatures: ['Color'] })).color;
printColorScheme(color);
```

Define the helper function `printColorScheme` to print the details of the color scheme to the console.

```
// Print a detected color scheme
function printColorScheme(colors) {
  console.log(`Image is in ${colors.isBwImg ? 'black and white' : 'color'}`);
  console.log(`Dominant colors: ${colors.dominantColors.join(', ')}`);
  console.log(`Dominant foreground color: ${colors.dominantColorForeground}`);
  console.log(`Dominant background color: ${colors.dominantColorBackground}`);
  console.log(`Suggested accent color: #${colors.accentColor}`);
}
```

Get domain-specific content

Image Analysis can use specialized model to do further analysis on images. See Domain-specific content for more details.

First, define the URL of an image to analyze:

```
const domainURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-sample-data-files/master/ComputerVision/Images/landmark.jpg';
```

The following code parses data about detected landmarks in the image.

```
// Analyze URL image
console.log('Analyzing image for landmarks...', domainURLImage.split('/').pop());
const domain = (await computerVisionClient.analyzeImageByDomain('landmarks', domainURLImage)).result.landmarks;

// Prints domain-specific, recognized objects
if (domain.length) {
  console.log(`${domain.length} ${domain.length == 1 ? 'landmark' : 'landmarks'} found:`);
  for (const obj of domain) {
    console.log(`    ${obj.name}`.padEnd(20) + `(${obj.confidence.toFixed(2)} confidence)`.padEnd(20) + `${formatRectDomain(obj.faceRectangle)}`);
  }
} else {
  console.log('No landmarks found.');
}
```

Define the helper function `formatRectDomain` to parse the location data about detected landmarks.

```
// Formats bounding box
function formatRectDomain(rect) {
  if (!rect) return '';
  return `top=${rect.top}`.padEnd(10) + `left=${rect.left}`.padEnd(10) + `bottom=${rect.top + rect.height}`.padEnd(12) +
    `right=${rect.left + rect.width}`.padEnd(10) + `(${rect.width}x${rect.height})`;
}
```

Get the image type

The following code prints information about the type of image—whether it is clip art or line drawing.

```
const typeURLImage = 'https://raw.githubusercontent.com/Azure-Samples/cognitive-services-python-sdk-samples/master/samples/vision/images/make_things_happen.jpg';

// Analyze URL image
console.log('Analyzing type in image...', typeURLImage.split('/').pop());
const types = (await computerVisionClient.analyzeImage(typeURLImage, { visualFeatures: ['ImageType']
})).imageType;
console.log(`Image appears to be ${describeType(types)}`);
```

Define the helper function `describeType`:

```
function describeType(imageType) {
  if (imageType.clipArtType && imageType.clipArtType > imageType.lineDrawingType) return 'clip art';
  if (imageType.lineDrawingType && imageType.clipArtType < imageType.lineDrawingType) return 'a line drawing';
  return 'a photograph';
}
```

I analyzed an image I ran into an issue

Run the application

Run the application with the `node` command on your quickstart file.

```
node index.js
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the Image Analysis client library and make basic image analysis calls. Next, learn more about the Analyze API features.

Call the Analyze API

- Image Analysis overview
- The source code for this sample can be found on GitHub.

Use the Image Analysis client library to analyze an image for tags, text description, faces, adult content, and more.

Reference documentation | Library source code | Package

Prerequisites

- An Azure subscription - Create one for free
- The latest version of Go
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Setting up

Create a Go project directory

In a console window (cmd, PowerShell, Terminal, Bash), create a new workspace for your Go project, named `my-app`, and navigate to it.

```
mkdir -p my-app/{src, bin, pkg}
cd my-app
```

Your workspace will contain three folders:

- **src** - This directory will contain source code and packages. Any packages installed with the `go get` command will go in this directory.
- **pkg** - This directory will contain the compiled Go package objects. These files all have an `.a` extension.
- **bin** - This directory will contain the binary executable files that are created when you run `go install`.

> **TIP**
> To learn more about the structure of a Go workspace, see the Go language documentation. This guide includes information for setting `$GOPATH` and `$GOROOT`.

Install the client library for Go

Next, install the client library for Go:

```
go get -u https://github.com/Azure/azure-sdk-for-
go/tree/master/services/cognitiveservices/v2.1/computervision
```

or if you use dep, within your repo run:

```
dep ensure -add https://github.com/Azure/azure-sdk-for-
go/tree/master/services/cognitiveservices/v2.1/computervision
```

Create a Go application

Next, create a file in the **src** directory named `sample-app.go`:

```
cd src
touch sample-app.go
```

Open `sample-app.go` in your preferred IDE or text editor. Then add the package name and import the following

libraries:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/Azure/azure-sdk-for-go/services/cognitiveservices/v2.0/computervision"
    "github.com/Azure/go-autorest/autorest"
    "io"
    "log"
    "os"
    "strings"
    "time"
)
```

Also, declare a context at the root of your script. You'll need this object to execute most Image Analysis function calls:

```
// Declare global so don't have to pass it to all of the tasks.
var computerVisionContext context.Context
```

Next, you'll begin adding code to carry out different Computer Vision operations.

I set up the client I ran into an issue

Object model

The following classes and interfaces handle some of the major features of the Image Analysis Go SDK.

NAME	DESCRIPTION
BaseClient	This class is needed for all Computer Vision functionality, such as image analysis and text reading. You instantiate it with your subscription information, and you use it to do most image operations.
ImageAnalysis	This type contains the results of an **AnalyzeImage** function call. There are similar types for each of the category-specific functions.
VisualFeatureTypes	This type defines the different kinds of image analysis that can be done in a standard Analyze operation. You specify a set of VisualFeatureTypes values depending on your needs.

Code examples

These code snippets show you how to do the following tasks with the Image Analysis client library for Go:

- Authenticate the client
- Analyze an image

Authenticate the client

> **NOTE**
>
> This step assumes you've created environment variables for your Computer Vision key and endpoint, named `COMPUTER_VISION_SUBSCRIPTION_KEY` and `COMPUTER_VISION_ENDPOINT` respectively.

Create a `main` function and add the following code to it to instantiate a client with your endpoint and key.

```
/*
 * Configure the Computer Vision client
 */
computerVisionClient := computervision.New(endpointURL);
computerVisionClient.Authorizer = autorest.NewCognitiveServicesAuthorizer(computerVisionKey)

computerVisionContext = context.Background()
/*
 * END - Configure the Computer Vision client
 */
```

I authenticated the client I ran into an issue

Analyze an image

The following code uses the client object to analyze a remote image and print the results to the console. You can get a text description, categorization, list of tags, detected objects, detected brands, detected faces, adult content flags, main colors, and image type.

Set up test image

First save a reference to the URL of the image you want to analyze. Put this inside your `main` function.

```
landmarkImageURL := "https://github.com/Azure-Samples/cognitive-services-sample-data-
files/raw/master/ComputerVision/Images/landmark.jpg"
```

> **TIP**
>
> You can also analyze a local image. See the BaseClient methods, such as **AnalyzeImageInStream**. Or, see the sample code on GitHub for scenarios involving local images.

Specify visual features

The following function calls extract different visual features from the sample image. You'll define these functions in the following sections.

```
// Analyze features of an image, remote
DescribeRemoteImage(computerVisionClient, landmarkImageURL)
CategorizeRemoteImage(computerVisionClient, landmarkImageURL)
TagRemoteImage(computerVisionClient, landmarkImageURL)
DetectFacesRemoteImage(computerVisionClient, facesImageURL)
DetectObjectsRemoteImage(computerVisionClient, objectsImageURL)
DetectBrandsRemoteImage(computerVisionClient, brandsImageURL)
DetectAdultOrRacyContentRemoteImage(computerVisionClient, adultRacyImageURL)
DetectColorSchemeRemoteImage(computerVisionClient, brandsImageURL)
DetectDomainSpecificContentRemoteImage(computerVisionClient, landmarkImageURL)
DetectImageTypesRemoteImage(computerVisionClient, detectTypeImageURL)
GenerateThumbnailRemoteImage(computerVisionClient, adultRacyImageURL)
```

Get image description

The following function gets the list of generated captions for the image. For more information about image description, see Describe images.

```
func DescribeRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("DESCRIBE IMAGE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    maxNumberDescriptionCandidates := new(int32)
    *maxNumberDescriptionCandidates = 1

    remoteImageDescription, err := client.DescribeImage(
            computerVisionContext,
            remoteImage,
            maxNumberDescriptionCandidates,
            "") // language
        if err != nil { log.Fatal(err) }

    fmt.Println("Captions from remote image: ")
    if len(*remoteImageDescription.Captions) == 0 {
        fmt.Println("No captions detected.")
    } else {
        for _, caption := range *remoteImageDescription.Captions {
            fmt.Printf("'%v' with confidence %.2f%%\n", *caption.Text, *caption.Confidence * 100)
        }
    }
    fmt.Println()
}
```

Get image category

The following function gets the detected category of the image. For more information, see Categorize images.

```
func CategorizeRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("CATEGORIZE IMAGE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesCategories}
    imageAnalysis, err := client.AnalyzeImage(
            computerVisionContext,
            remoteImage,
            features,
            []computervision.Details{},
            "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Categories from remote image: ")
    if len(*imageAnalysis.Categories) == 0 {
        fmt.Println("No categories detected.")
    } else {
        for _, category := range *imageAnalysis.Categories {
            fmt.Printf("'%v' with confidence %.2f%%\n", *category.Name, *category.Score * 100)
        }
    }
    fmt.Println()
}
```

Get image tags

The following function gets the set of detected tags in the image. For more information, see Content tags.

```
func TagRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----------------------------------------")
    fmt.Println("TAG IMAGE - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    remoteImageTags, err := client.TagImage(
            computerVisionContext,
            remoteImage,
            "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Tags in the remote image: ")
    if len(*remoteImageTags.Tags) == 0 {
        fmt.Println("No tags detected.")
    } else {
        for _, tag := range *remoteImageTags.Tags {
            fmt.Printf("'%v' with confidence %.2f%%\n", *tag.Name, *tag.Confidence * 100)
        }
    }
    fmt.Println()
}
```

Detect objects

The following function detects common objects in the image and prints them to the console. For more information, see Object detection.

```
func DetectObjectsRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----------------------------------------")
    fmt.Println("DETECT OBJECTS - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    imageAnalysis, err := client.DetectObjects(
            computerVisionContext,
            remoteImage,
    )
    if err != nil { log.Fatal(err) }

    fmt.Println("Detecting objects in remote image: ")
    if len(*imageAnalysis.Objects) == 0 {
        fmt.Println("No objects detected.")
    } else {
        // Print the objects found with confidence level and bounding box locations.
        for _, object := range *imageAnalysis.Objects {
            fmt.Printf("'%v' with confidence %.2f%% at location (%v, %v), (%v, %v)\n",
                *object.Object, *object.Confidence * 100,
                *object.Rectangle.X, *object.Rectangle.X + *object.Rectangle.W,
                *object.Rectangle.Y, *object.Rectangle.Y + *object.Rectangle.H)
        }
    }
    fmt.Println()
}
```

Detect brands

The following code detects corporate brands and logos in the image and prints them to the console. For more information, Brand detection.

First, declare a reference to a new image within your `main` function.

```
brandsImageURL := "https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/images/gray-shirt-logo.jpg"
```

The following code defines the brand detection function.

```
func DetectBrandsRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("DETECT BRANDS - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the kinds of features you want returned.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesBrands}

    imageAnalysis, err := client.AnalyzeImage(
        computerVisionContext,
        remoteImage,
        features,
        []computervision.Details{},
        "en")
    if err != nil { log.Fatal(err) }

    fmt.Println("Detecting brands in remote image: ")
    if len(*imageAnalysis.Brands) == 0 {
        fmt.Println("No brands detected.")
    } else {
        // Get bounding box around the brand and confidence level it's correctly identified.
        for _, brand := range *imageAnalysis.Brands {
            fmt.Printf("'%v' with confidence %.2f%% at location (%v, %v), (%v, %v)\n",
                *brand.Name, *brand.Confidence * 100,
                *brand.Rectangle.X, *brand.Rectangle.X + *brand.Rectangle.W,
                *brand.Rectangle.Y, *brand.Rectangle.Y + *brand.Rectangle.H)
        }
    }
    fmt.Println()
}
```

Detect faces

The following function returns the detected faces in the image with their rectangle coordinates and certain face attributes. For more information, see Face detection.

```
func DetectFacesRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("DETECT FACES - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the features you want returned with the API call.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesFaces}
    imageAnalysis, err := client.AnalyzeImage(
            computerVisionContext,
            remoteImage,
            features,
            []computervision.Details{},
            "")
        if err != nil { log.Fatal(err) }

    fmt.Println("Detecting faces in a remote image ...")
    if len(*imageAnalysis.Faces) == 0 {
        fmt.Println("No faces detected.")
    } else {
        // Print the bounding box locations of the found faces.
        for _, face := range *imageAnalysis.Faces {
            fmt.Printf("'%v' of age %v at location (%v, %v), (%v, %v)\n",
                face.Gender, *face.Age,
                *face.FaceRectangle.Left, *face.FaceRectangle.Top,
                *face.FaceRectangle.Left + *face.FaceRectangle.Width,
                *face.FaceRectangle.Top + *face.FaceRectangle.Height)
        }
    }
    fmt.Println()
}
```

Detect adult, racy, or gory content

The following function prints the detected presence of adult content in the image. For more information, see Adult, racy, gory content.

```
func DetectAdultOrRacyContentRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("DETECT ADULT OR RACY CONTENT - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the features you want returned from the API call.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesAdult}
    imageAnalysis, err := client.AnalyzeImage(
            computerVisionContext,
            remoteImage,
            features,
            []computervision.Details{},
            "") // language, English is default
    if err != nil { log.Fatal(err) }

    // Print whether or not there is questionable content.
    // Confidence levels: low means content is OK, high means it's not.
    fmt.Println("Analyzing remote image for adult or racy content: ");
    fmt.Printf("Is adult content: %v with confidence %.2f%%\n", *imageAnalysis.Adult.IsAdultContent,
*imageAnalysis.Adult.AdultScore * 100)
    fmt.Printf("Has racy content: %v with confidence %.2f%%\n", *imageAnalysis.Adult.IsRacyContent,
*imageAnalysis.Adult.RacyScore * 100)
    fmt.Println()
}
```

Get image color scheme

The following function prints the detected color attributes in the image, like the dominant colors and accent color. For more information, see Color schemes.

```
func DetectColorSchemeRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----------------------------------------")
    fmt.Println("DETECT COLOR SCHEME - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    // Define the features you'd like returned with the result.
    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesColor}
    imageAnalysis, err := client.AnalyzeImage(
            computerVisionContext,
            remoteImage,
            features,
            []computervision.Details{},
            "") // language, English is default
    if err != nil { log.Fatal(err) }

    fmt.Println("Color scheme of the remote image: ");
    fmt.Printf("Is black and white: %v\n", *imageAnalysis.Color.IsBWImg)
    fmt.Printf("Accent color: 0x%v\n", *imageAnalysis.Color.AccentColor)
    fmt.Printf("Dominant background color: %v\n", *imageAnalysis.Color.DominantColorBackground)
    fmt.Printf("Dominant foreground color: %v\n", *imageAnalysis.Color.DominantColorForeground)
    fmt.Printf("Dominant colors: %v\n", strings.Join(*imageAnalysis.Color.DominantColors, ", "))
    fmt.Println()
}
```

Get domain-specific content

Image Analysis can use specialized models to do further analysis on images. For more information, see Domain-specific content.

The following code parses data about detected celebrities in the image.

```
func DetectDomainSpecificContentRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("-----------------------------------------")
    fmt.Println("DETECT DOMAIN-SPECIFIC CONTENT - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    fmt.Println("Detecting domain-specific content in the local image ...")

    // Check if there are any celebrities in the image.
    celebrities, err := client.AnalyzeImageByDomain(
            computerVisionContext,
            "celebrities",
            remoteImage,
            "") // language, English is default
    if err != nil { log.Fatal(err) }

    fmt.Println("\nCelebrities: ")

    // Marshal the output from AnalyzeImageByDomain into JSON.
    data, err := json.MarshalIndent(celebrities.Result, "", "\t")

    // Define structs for which to unmarshal the JSON.
    type Celebrities struct {
        Name string `json:"name"`
    }

    type CelebrityResult struct {
        Celebrities []Celebrities `json:"celebrities"`
    }

    var celebrityResult CelebrityResult

    // Unmarshal the data.
    err = json.Unmarshal(data, &celebrityResult)
    if err != nil { log.Fatal(err) }

    // Check if any celebrities detected.
    if len(celebrityResult.Celebrities) == 0 {
        fmt.Println("No celebrities detected.")
    } else {
        for _, celebrity := range celebrityResult.Celebrities {
            fmt.Printf("name: %v\n", celebrity.Name)
        }
    }
```

The following code parses data about detected landmarks in the image.

```
    fmt.Println("\nLandmarks: ")

    // Check if there are any landmarks in the image.
    landmarks, err := client.AnalyzeImageByDomain(
            computerVisionContext,
            "landmarks",
            remoteImage,
            "")
    if err != nil { log.Fatal(err) }

    // Marshal the output from AnalyzeImageByDomain into JSON.
    data, err = json.MarshalIndent(landmarks.Result, "", "\t")

    // Define structs for which to unmarshal the JSON.
    type Landmarks struct {
        Name string `json:"name"`
    }

    type LandmarkResult struct {
        Landmarks []Landmarks `json:"landmarks"`
    }

    var landmarkResult LandmarkResult

    // Unmarshal the data.
    err = json.Unmarshal(data, &landmarkResult)
    if err != nil { log.Fatal(err) }

    // Check if any celebrities detected.
    if len(landmarkResult.Landmarks) == 0 {
        fmt.Println("No landmarks detected.")
    } else {
        for _, landmark := range landmarkResult.Landmarks {
            fmt.Printf("name: %v\n", landmark.Name)
        }
    }
    fmt.Println()
}
```

Get the image type

The following function prints information about the type of image—whether it's clip art or a line drawing.

```
func DetectImageTypesRemoteImage(client computervision.BaseClient, remoteImageURL string) {
    fmt.Println("----------------------------------------")
    fmt.Println("DETECT IMAGE TYPES - remote")
    fmt.Println()
    var remoteImage computervision.ImageURL
    remoteImage.URL = &remoteImageURL

    features := []computervision.VisualFeatureTypes{computervision.VisualFeatureTypesImageType}

    imageAnalysis, err := client.AnalyzeImage(
            computerVisionContext,
            remoteImage,
            features,
            []computervision.Details{},
            "")
    if err != nil { log.Fatal(err) }

    fmt.Println("Image type of remote image:")

    fmt.Println("\nClip art type: ")
    switch *imageAnalysis.ImageType.ClipArtType {
    case 0:
        fmt.Println("Image is not clip art.")
    case 1:
        fmt.Println("Image is ambiguously clip art.")
    case 2:
        fmt.Println("Image is normal clip art.")
    case 3:
        fmt.Println("Image is good clip art.")
    }

    fmt.Println("\nLine drawing type: ")
    if *imageAnalysis.ImageType.LineDrawingType == 1 {
        fmt.Println("Image is a line drawing.")
    } else {
        fmt.Println("Image is not a line drawing.")
    }
    fmt.Println()
}
```

I analyzed an image I ran into an issue

Run the application

Run the application from your application directory with the `go run` command.

```
go run sample-app.go
```

I ran the application I ran into an issue

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

I cleaned up resources I ran into an issue

Next steps

In this quickstart, you learned how to install the Image Analysis client library and make basic image analysis calls. Next, learn more about the Analyze API features.

Call the Analyze API

- Image Analysis overview
- The source code for this sample can be found on GitHub.

Use the Image Analysis REST API to:

- Analyze an image for tags, text description, faces, adult content, and more.
- Generate a thumbnail with smart cropping

> **NOTE**
> This quickstart uses cURL commands to call the REST API. You can also call the REST API using a programming language. See the GitHub samples for examples in C#, Python, Java, JavaScript, and Go.

Prerequisites

- An Azure subscription - Create one for free
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.
- cURL installed

Analyze an image

To analyze an image for a variety of visual features, do the following steps:

1. Copy the following command into a text editor.
2. Make the following changes in the command where needed:
   a. Replace the value of `<subscriptionKey>` with your subscription key.
   b. Replace the first part of the request URL (`westcentralus`) with the text in your own endpoint URL.

      > **NOTE**
      > New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

   c. Optionally, change the image URL in the request body (`http://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg\`) to the URL of a different image to be analyzed.
3. Open a command prompt window.
4. Paste the command from the text editor into the command prompt window, and then run the command.

```
curl -H "Ocp-Apim-Subscription-Key: <subscriptionKey>" -H "Content-Type: application/json"
"https://westcentralus.api.cognitive.microsoft.com/vision/v3.2/analyze?
visualFeatures=Categories,Description&details=Landmarks" -d "
{\"url\":\"http://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg\"}"
```

Examine the response

A successful response is returned in JSON. The sample application parses and displays a successful response in the command prompt window, similar to the following example:


```
{
  "categories": [
    {
      "name": "outdoor_water",
      "score": 0.9921875,
      "detail": {
        "landmarks": []
      }
    }
  ],
  "description": {
    "tags": [
      "nature",
      "water",
      "waterfall",
      "outdoor",
      "rock",
      "mountain",
      "rocky",
      "grass",
      "hill",
      "covered",
      "hillside",
      "standing",
      "side",
      "group",
      "walking",
      "white",
      "man",
      "large",
      "snow",
      "grazing",
      "forest",
      "slope",
      "herd",
      "river",
      "giraffe",
      "field"
    ],
    "captions": [
      {
        "text": "a large waterfall over a rocky cliff",
        "confidence": 0.916458423253597
      }
    ]
  },
  "requestId": "b6e33879-abb2-43a0-a96e-02cb5ae0b795",
  "metadata": {
    "height": 959,
    "width": 1280,
    "format": "Jpeg"
  }
}
```

Generate a thumbnail

You can use Image Analysis to generate a thumbnail with smart cropping. You specify the desired height and width, which can differ in aspect ratio from the input image. Image Analysis uses smart cropping to intelligently identify the area of interest and generate cropping coordinates around that region.

To create and run the sample, do the following steps:

1. Copy the following command into a text editor.
2. Make the following changes in the command where needed:
   a. Replace the value of `<subscriptionKey>` with your subscription key.
   b. Replace the value of `<thumbnailFile>` with the path and name of the file in which to save the returned thumbnail image.
   c. Replace the first part of the request URL (`westcentralus`) with the text in your own endpoint URL.

      > **NOTE**
      > New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

   d. Optionally, change the image URL in the request body (`https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Shorkie_Poo_Puppy.jpg/1280px-Shorkie_Poo_Puppy.jpg\`) to the URL of a different image from which to generate a thumbnail.
3. Open a command prompt window.
4. Paste the command from the text editor into the command prompt window.
5. Press enter to run the program.

```
curl -H "Ocp-Apim-Subscription-Key: <subscriptionKey>" -o <thumbnailFile> -H "Content-Type: application/json" "https://westus.api.cognitive.microsoft.com/vision/v3.2/generateThumbnail?width=100&height=100&smartCropping=true" -d "{\"url\":\"https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Shorkie_Poo_Puppy.jpg/1280px-Shorkie_Poo_Puppy.jpg\"}"
```

Examine the response

A successful response writes the thumbnail image to the file specified in `<thumbnailFile>`. If the request fails, the response contains an error code and a message to help determine what went wrong. If the request seems to succeed but the created thumbnail is not a valid image file, it might be that your subscription key is not valid.

Next steps

In this quickstart, you learned how to install make basic image analysis calls using the REST API. Next, learn more about the Analyze API features.

Call the Analyze API

- Image Analysis overview

Sample: Explore an image processing app with C#

5/25/2021 • 19 minutes to read • Edit Online

Explore a basic Windows application that uses Computer Vision to perform optical character recognition (OCR), create smart-cropped thumbnails, plus detect, categorize, tag and describe visual features, including faces, in an image. The below example lets you submit an image URL or a locally stored file. You can use this open source example as a template for building your own app for Windows using the Computer Vision API and Windows Presentation Foundation (WPF), a part of .NET Framework.

- Get the sample app from GitHub
- Open and build the sample app in Visual Studio
- Run the sample app and interact with it to perform various scenarios
- Explore the various scenarios included with the sample app

Prerequisites

Before exploring the sample app, ensure that you've met the following prerequisites:

- You must have Visual Studio 2015 or later.
- An Azure subscription - Create one for free
- Once you have your Azure subscription, create a Computer Vision resource in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to connect your application to the Computer Vision service. You'll paste your key and endpoint into the code below later in the quickstart.
  - You can use the free pricing tier (`F0`) to try the service, and upgrade later to a paid tier for production.

Get the sample app

The Computer Vision sample app is available on GitHub from the Microsoft/Cognitive-Vision-Windows repository. This repository also includes the `Microsoft/Cognitive-Common-Windows` repository as a Git submodule. You can recursively clone this repository, including the submodule, either by using the `git clone --recurse-submodules` command from the command line, or by using GitHub Desktop.

For example, to recursively clone the repository for the Computer Vision sample app from a command prompt, run the following command:

```
git clone --recurse-submodules https://github.com/Microsoft/Cognitive-Vision-Windows.git
```

> **IMPORTANT**
> Do not download this repository as a ZIP. Git doesn't include submodules when downloading a repository as a ZIP.

Get optional sample images

You can optionally use the sample images included with the Face sample app, available on GitHub from the `Microsoft/Cognitive-Face-Windows` repository. That sample app includes a folder, `/Data` , which contains multiple images of people. You can recursively clone this repository, as well, by the methods described for the Computer Vision sample app.

For example, to recursively clone the repository for the Face sample app from a command prompt, run the following command:

```
git clone --recurse-submodules https://github.com/Microsoft/Cognitive-Face-Windows.git
```

Open and build the sample app in Visual Studio

You must build the sample app first, so that Visual Studio can resolve dependencies, before you can run or explore the sample app. To open and build the sample app, do the following steps:

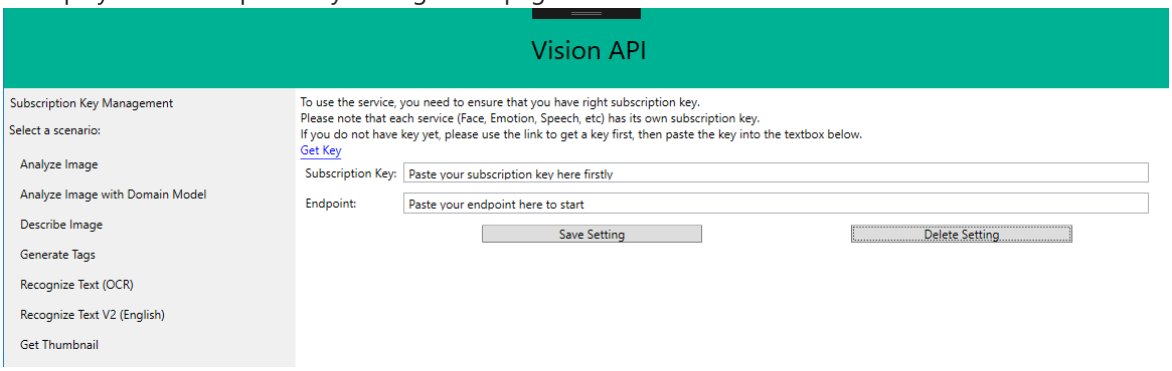
1. Open the Visual Studio solution file, `/Sample-WPF/VisionAPI-WPF-Samples.sln`, in Visual Studio.
2. Ensure that the Visual Studio solution contains two projects:
   - SampleUserControlLibrary
   - VisionAPI-WPF-Samples

   If the SampleUserControlLibrary project is unavailable, confirm that you've recursively cloned the `Microsoft/Cognitive-Vision-Windows` repository.
3. In Visual Studio, either press Ctrl+Shift+B or choose **Build** from the ribbon menu and then choose **Build Solution** to build the solution.

Run and interact with the sample app

You can run the sample app, to see how it interacts with you and with the Computer Vision client library when performing various tasks, such as generating thumbnails or tagging images. To run and interact with the sample app, do the following steps:

1. After the build is complete, either press **F5** or choose **Debug** from the ribbon menu and then choose **Start debugging** to run the sample app.
2. When the sample app is displayed, choose **Subscription Key Management** from the navigation pane to display the Subscription Key Management page.



3. Enter your subscription key in **Subscription Key**.
4. Enter the endpoint URL in **Endpoint**.

   > **NOTE**
   > New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

5. If you don't want to enter your subscription key and endpoint URL the next time you run the sample app, choose **Save Setting** to save the subscription key and endpoint URL to your computer. If you want to

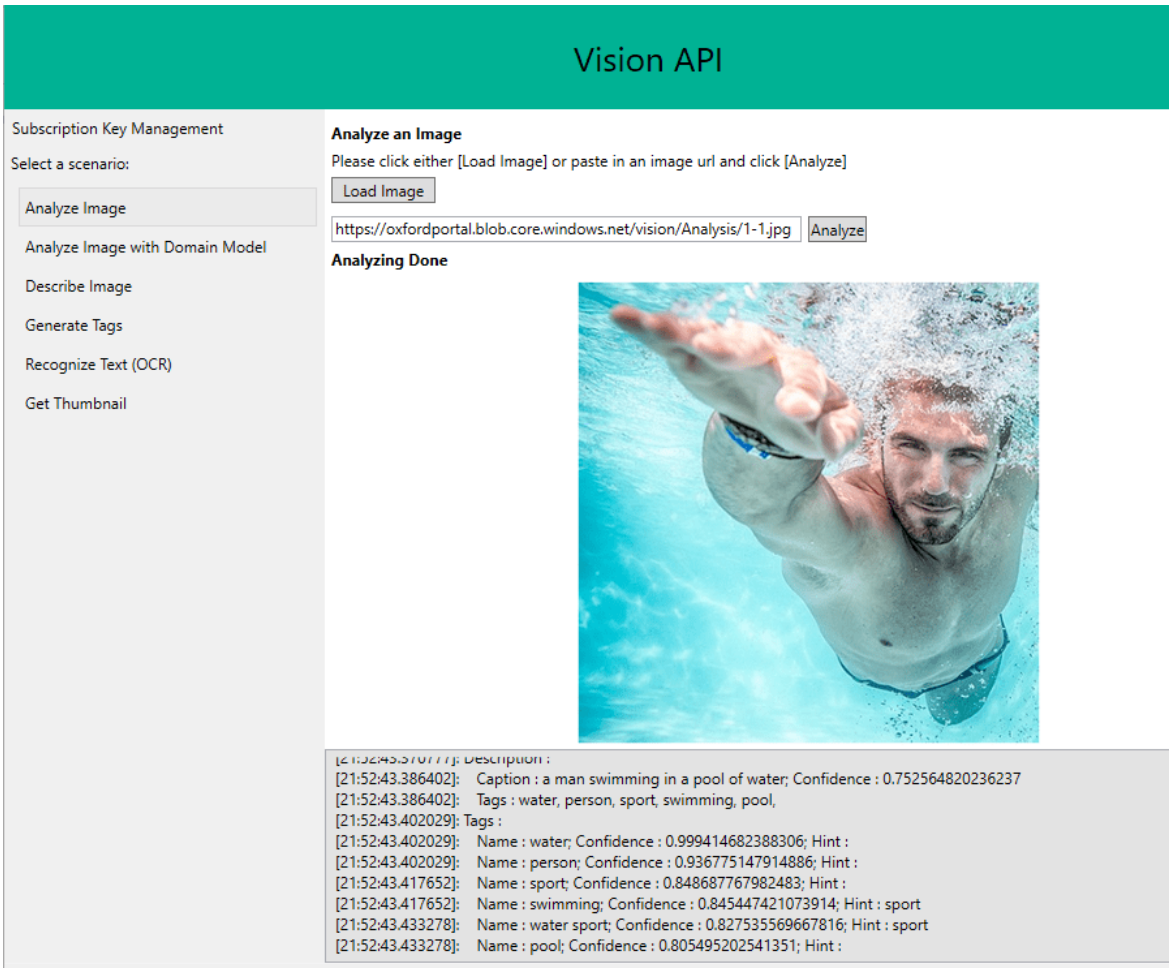
delete your previously-saved subscription key and endpoint URL, choose **Delete Setting**.

> **NOTE**
> The sample app uses isolated storage, and `System.IO.IsolatedStorage`, to store your subscription key and endpoint URL.

6. Under **Select a scenario** in the navigation pane, select one of the scenarios currently included with the sample app:

SCENARIO	DESCRIPTION
Analyze Image	Uses the Analyze Image operation to analyze a local or remote image. You can choose the visual features and language for the analysis, and see both the image and the results.
Analyze Image with Domain Model	Uses the List Domain Specific Models operation to list the domain models from which you can select, and the Recognize Domain Specific Content operation to analyze a local or remote image using the selected domain model. You can also choose the language for the analysis.
Describe Image	Uses the Describe Image operation to create a human-readable description of a local or remote image. You can also choose the language for the description.
Generate Tags	Uses the Tag Image operation to tag the visual features of a local or remote image. You can also choose the language used for the tags.
Recognize Text (OCR)	Uses the OCR operation to recognize and extract printed text from an image. You can either choose the language to use, or let Computer Vision auto-detect the language.
Recognize Text V2 (English)	Uses the Recognize Text and Get Recognize Text Operation Result operations to asynchronously recognize and extract printed or handwritten text from an image.
Get Thumbnail	Uses the Get Thumbnail operation to generate a thumbnail for a local or remote image.

The following screenshot illustrates the page provided for the Analyze Image scenario, after analyzing a sample image.



Explore the sample app

The Visual Studio solution for the Computer Vision sample app contains two projects:

- SampleUserControlLibrary
  The SampleUserControlLibrary project provides functionality shared by multiple Cognitive Services samples. The project contains the following:
  - SampleScenarios
    A UserControl that provides a standardized presentation, such as the title bar, navigation pane, and content pane, for samples. The Computer Vision sample app uses this control in the MainWindow.xaml window to display scenario pages and access information shared across scenarios, such as the subscription key and endpoint URL.
  - SubscriptionKeyPage
    A Page that provides a standardized layout for entering a subscription key and endpoint URL for the sample app. The Computer Vision sample app uses this page to manage the subscription key and endpoint URL used by the scenario pages.
  - VideoResultControl
    A UserControl that provides a standardized presentation for video information. The Computer Vision sample app doesn't use this control.
- VisionAPI-WPF-Samples
  The main project for the Computer Vision sample app, this project contains all of the interesting functionality for Computer Vision. The project contains the following:
  - AnalyzeInDomainPage.xaml
    The scenario page for the Analyze Image with Domain Model scenario.
  - AnalyzeImage.xaml
    The scenario page for the Analyze Image scenario.

- DescribePage.xaml
  The scenario page for the Describe Image scenario.
- ImageScenarioPage.cs
  The ImageScenarioPage class, from which all of the scenario pages in the sample app are derived. This class manages functionality, such as providing credentials and formatting output, shared by all of the scenario pages.
- MainWindow.xaml
  The main window for the sample app, it uses the SampleScenarios control to present the SubscriptionKeyPage and scenario pages.
- OCRPage.xaml
  The scenario page for the Recognize Text (OCR) scenario.
- RecognizeLanguage.cs
  The RecognizeLanguage class, which provides information about the languages supported by the various methods in the sample app.
- TagsPage.xaml
  The scenario page for the Generate Tags scenario.
- TextRecognitionPage.xaml
  The scenario page for the Recognize Text V2 (English) scenario.
- ThumbnailPage.xaml
  The scenario page for the Get Thumbnail scenario.

Explore the sample code

Key portions of sample code are framed with comment blocks that start with `KEY SAMPLE CODE STARTS HERE` and end with `KEY SAMPLE CODE ENDS HERE`, to make it easier for you to explore the sample app. These key portions of sample code contain the code most relevant to learning how to use the Computer Vision API client library to do various tasks. You can search for `KEY SAMPLE CODE STARTS HERE` in Visual Studio to move between the most relevant sections of code in the Computer Vision sample app.

For example, the `UploadAndAnalyzeImageAsync` method, shown following and included in AnalyzePage.xaml, demonstrates how to use the client library to analyze a local image by invoking the `ComputerVisionClient.AnalyzeImageInStreamAsync` method.

```
private async Task<ImageAnalysis> UploadAndAnalyzeImageAsync(string imageFilePath)
{
    // -----------------------------------------------------------------------
    // KEY SAMPLE CODE STARTS HERE
    // -----------------------------------------------------------------------

    //
    // Create Cognitive Services Vision API Service client.
    //
    using (var client = new ComputerVisionClient(Credentials) { Endpoint = Endpoint })
    {
        Log("ComputerVisionClient is created");

        using (Stream imageFileStream = File.OpenRead(imageFilePath))
        {
            //
            // Analyze the image for all visual features.
            //
            Log("Calling ComputerVisionClient.AnalyzeImageInStreamAsync()...");
            VisualFeatureTypes[] visualFeatures = GetSelectedVisualFeatures();
            string language = (_language.SelectedItem as RecognizeLanguage).ShortCode;
            ImageAnalysis analysisResult = await client.AnalyzeImageInStreamAsync(imageFileStream,
visualFeatures, null, language);
            return analysisResult;
        }
    }

    // -----------------------------------------------------------------------
    // KEY SAMPLE CODE ENDS HERE
    // -----------------------------------------------------------------------
}
```

Explore the client library

This sample app uses the Computer Vision API client library, a thin C# client wrapper for the Computer Vision API in Azure Cognitive Services. The client library is available from NuGet in the Microsoft.Azure.CognitiveServices.Vision.ComputerVision package. When you built the Visual Studio application, you retrieved the client library from its corresponding NuGet package. You can also view the source code for the client library in the `/ClientLibrary` folder of the `Microsoft/Cognitive-Vision-Windows` repository.

The client library's functionality centers around the `ComputerVisionClient` class, in the `Microsoft.Azure.CognitiveServices.Vision.ComputerVision` namespace, while the models used by the `ComputerVisionClient` class when interacting with Computer Vision are found in the `Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models` namespace. In the various XAML scenario pages included with the sample app, you'll find the following `using` directives for those namespaces:

```
// -----------------------------------------------------------------------
// KEY SAMPLE CODE STARTS HERE
// Use the following namespace for ComputerVisionClient.
// -----------------------------------------------------------------------
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
// -----------------------------------------------------------------------
// KEY SAMPLE CODE ENDS HERE
// -----------------------------------------------------------------------
```

You'll learn more about the various methods included with the `ComputerVisionClient` class as you explore the scenarios included with the Computer Vision sample app.

Explore the Analyze Image scenario

This scenario is managed by the AnalyzePage.xaml page. You can choose the visual features and language for the analysis, and see both the image and the results. The scenario page does this by using one of the following methods, depending on the source of the image:

- UploadAndAnalyzeImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageInStreamAsync` method.
- AnalyzeUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageAsync` method.

The `UploadAndAnalyzeImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the visual features and language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageInStreamAsync` method, passing the `Stream` for the file, the visual features, and the language, then returns the result as an `ImageAnalysis` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `AnalyzeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the visual features and language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageInStreamAsync` method, passing the image URL, the visual features, and the language, then returns the result as an `ImageAnalysis` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Analyze Image with Domain Model scenario

This scenario is managed by the AnalyzeInDomainPage.xaml page. You can choose a domain model, such as `celebrities` or `landmarks`, and language to perform a domain-specific analysis of the image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- GetAvailableDomainModelsAsync
  This method gets the list of available domain models from Computer Vision and populates the `_domainModelComboBox` ComboBox control on the page, using the `ComputerVisionClient.ListModelsAsync` method.
- UploadAndAnalyzeInDomainImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageByDomainInStreamAsync` method.
- AnalyzeInDomainUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.AnalyzeImageByDomainAsync` method.

The `UploadAndAnalyzeInDomainImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageByDomainInStreamAsync` method, passing the `Stream` for the file, the name of the domain model, and the language, then returns the result as an `DomainModelResults` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `AnalyzeInDomainUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.AnalyzeImageByDomainAsync` method, passing the image URL, the visual features, and the

language, then returns the result as an `DomainModelResults` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Describe Image scenario

This scenario is managed by the DescribePage.xaml page. You can choose a language to create a human-readable description of the image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- UploadAndDescribeImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.DescribeImageInStreamAsync` method.
- DescribeUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.DescribeImageAsync` method.

The `UploadAndDescribeImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.DescribeImageInStreamAsync` method, passing the `Stream` for the file, the maximum number of candidates (in this case, 3), and the language, then returns the result as an `ImageDescription` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `DescribeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.DescribeImageAsync` method, passing the image URL, the maximum number of candidates (in this case, 3), and the language, then returns the result as an `ImageDescription` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Generate Tags scenario

This scenario is managed by the TagsPage.xaml page. You can choose a language to tag the visual features of an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- UploadAndGetTagsForImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.TagImageInStreamAsync` method.
- GenerateTagsForUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.TagImageAsync` method.

The `UploadAndGetTagsForImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.TagImageInStreamAsync` method, passing the `Stream` for the file and the language, then returns the result as a `TagResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `GenerateTagsForUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the

`ComputerVisionClient.TagImageAsync` method, passing the image URL and the language, then returns the result as a `TagResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Recognize Text (OCR) scenario

This scenario is managed by the OCRPage.xaml page. You can choose a language to recognize and extract printed text from an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- UploadAndRecognizeImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.RecognizePrintedTextInStreamAsync` method.
- RecognizeUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.RecognizePrintedTextAsync` method.

The `UploadAndRecognizeImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`, then gets the language selected in the scenario page. It calls the `ComputerVisionClient.RecognizePrintedTextInStreamAsync` method, indicating that orientation is not detected and passing the `Stream` for the file and the language, then returns the result as an `OcrResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `RecognizeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It gets the language selected in the scenario page. It calls the `ComputerVisionClient.RecognizePrintedTextAsync` method, indicating that orientation is not detected and passing the image URL and the language, then returns the result as an `OcrResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Explore the Recognize Text V2 (English) scenario

This scenario is managed by the TextRecognitionPage.xaml page. You can choose the recognition mode and a language to asynchronously recognize and extract either printed or handwritten text from an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- UploadAndRecognizeImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `RecognizeAsync` method and passing a parameterized delegate for the `ComputerVisionClient.RecognizeTextInStreamAsync` method.
- RecognizeUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `RecognizeAsync` method and passing a parameterized delegate for the `ComputerVisionClient.RecognizeTextAsync` method.
- RecognizeAsync This method handles the asynchronous calling for both the `UploadAndRecognizeImageAsync` and `RecognizeUrlAsync` methods, as well as polling for results by calling the `ComputerVisionClient.GetTextOperationResultAsync` method.

Unlike the other scenarios included in the Computer Vision sample app, this scenario is asynchronous, in that one method is called to start the process, but a different method is called to check on the status and return the results of that process. The logical flow in this scenario is somewhat different from that in the other scenarios.

The `UploadAndRecognizeImageAsync` method opens the local file specified in `imageFilePath` for reading as a `Stream`, then calls the `RecognizeAsync` method, passing:

- A lambda expression for a parameterized asynchronous delegate of the `ComputerVisionClient.RecognizeTextInStreamAsync` method, with the `Stream` for the file and the recognition mode as parameters, in `GetHeadersAsyncFunc`.
- A lambda expression for a delegate to get the `Operation-Location` response header value, in `GetOperationUrlFunc`.

The `RecognizeUrlAsync` method calls the `RecognizeAsync` method, passing:

- A lambda expression for a parameterized asynchronous delegate of the `ComputerVisionClient.RecognizeTextAsync` method, with the URL of the remote image and the recognition mode as parameters, in `GetHeadersAsyncFunc`.
- A lambda expression for a delegate to get the `Operation-Location` response header value, in `GetOperationUrlFunc`.

When the `RecognizeAsync` method is completed, both `UploadAndRecognizeImageAsync` and `RecognizeUrlAsync` methods return the result as a `TextOperationResult` instance. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `RecognizeAsync` method calls the parameterized delegate for either the `ComputerVisionClient.RecognizeTextInStreamAsync` or `ComputerVisionClient.RecognizeTextAsync` method passed in `GetHeadersAsyncFunc` and waits for the response. The method then calls the delegate passed in `GetOperationUrlFunc` to get the `Operation-Location` response header value from the response. This value is the URL used to retrieve the results of the method passed in `GetHeadersAsyncFunc` from Computer Vision.

The `RecognizeAsync` method then calls the `ComputerVisionClient.GetTextOperationResultAsync` method, passing the URL retrieved from the `Operation-Location` response header, to get the status and result of the method passed in `GetHeadersAsyncFunc`. If the status doesn't indicate that the method completed, successfully or unsuccessfully, the `RecognizeAsync` method calls `ComputerVisionClient.GetTextOperationResultAsync` 3 more times, waiting 3 seconds between calls. The `RecognizeAsync` method returns the results to the method that called it.

Explore the Get Thumbnail scenario

This scenario is managed by the ThumbnailPage.xaml page. You can indicate whether to use smart cropping, and specify desired height and width, to generate a thumbnail from an image, and see both the image and the results. The scenario page uses the following methods, depending on the source of the image:

- UploadAndThumbnailImageAsync
  This method is used for local images, in which the image must be encoded as a `Stream` and sent to Computer Vision by calling the `ComputerVisionClient.GenerateThumbnailInStreamAsync` method.
- ThumbnailUrlAsync
  This method is used for remote images, in which the URL for the image is sent to Computer Vision by calling the `ComputerVisionClient.GenerateThumbnailAsync` method.

The `UploadAndThumbnailImageAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. Because the sample app is analyzing a local image, it has to send the contents of that image to Computer Vision. It opens the local file specified in `imageFilePath` for reading as a `Stream`. It calls the `ComputerVisionClient.GenerateThumbnailInStreamAsync` method, passing the width, height, the `Stream` for the file, and whether to use smart cropping, then returns the result as a `Stream`. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

The `RecognizeUrlAsync` method creates a new `ComputerVisionClient` instance, using the specified subscription key and endpoint URL. It calls the `ComputerVisionClient.GenerateThumbnailAsync` method, passing the width, height, the URL for the image, and whether to use smart cropping, then returns the result as a `Stream`. The methods inherited from the `ImageScenarioPage` class present the returned results in the scenario page.

Clean up resources

When no longer needed, delete the folder into which you cloned the `Microsoft/Cognitive-Vision-Windows` repository. If you opted to use the sample images, also delete the folder into which you cloned the `Microsoft/Cognitive-Face-Windows` repository.

Next steps

Get started with Face service

Call the Image Analysis API

5/25/2021 • 3 minutes to read • Edit Online

This article demonstrates how to call the Image Analysis API to return information about an image's visual features.

This guide assumes you have already create a Computer Vision resource and obtained a subscription key and endpoint URL. If you haven't, follow a quickstart to get started.

Submit data to the service

You submit either a local image or a remote image to the Analyze API. For local, you put the binary image data in the HTTP request body. For remote, you specify the image's URL by formatting the request body like the following: `{"url":"http://example.com/images/test.jpg"}`.

Determine how to process the data

Select visual features

The Analyze API gives you access to all of the service's image analysis features. You need to specify which features you want to use by setting the URL query parameters. A parameter can have multiple values, separated by commas. Each feature you specify will require additional computation time, so only specify what you need.

URL PARAMETER	VALUE	DESCRIPTION
`visualFeatures`	`Adult`	detects if the image is pornographic in nature (depicts nudity or a sex act), or is gory (depicts extreme violence or blood). Sexually suggestive content (aka racy content) is also detected.
	`Brands`	detects various brands within an image, including the approximate location. The Brands argument is only available in English.
	`Categories`	categorizes image content according to a taxonomy defined in documentation. This is the default value of `visualFeatures`.
	`Color`	determines the accent color, dominant color, and whether an image is black&white.
	`Description`	describes the image content with a complete sentence in supported languages.
	`Faces`	detects if faces are present. If present, generate coordinates, gender and age.

URL PARAMETER	VALUE	DESCRIPTION
	`ImageType`	detects if image is clip art or a line drawing.
	`Objects`	detects various objects within an image, including the approximate location. The Objects argument is only available in English.
	`Tags`	tags the image with a detailed list of words related to the image content.
`details`	`Celebrities`	identifies celebrities if detected in the image.
	`Landmarks`	identifies landmarks if detected in the image.

A populated URL might look like the following:

```
https://{endpoint}/vision/v2.1/analyze?visualFeatures=Description,Tags&details=Celebrities
```

Specify languages

You can also specify the language of the returned data. The following URL query parameter specifies the language. The default value is `en`.

URL PARAMETER	VALUE	DESCRIPTION
`language`	`en`	English
	`es`	Spanish
	`ja`	Japanese
	`pt`	Portuguese
	`zh`	Simplified Chinese

A populated URL might look like the following:

```
https://{endpoint}/vision/v2.1/analyze?visualFeatures=Description,Tags&details=Celebrities&language=en
```

> **NOTE**
>
> **Scoped API calls**
>
> Some of the features in Image Analysis can be called directly as well as through the Analyze API call. For example, you can do a scoped analysis of only image tags by making a request to `https://{endpoint}/vision/v3.2/tag`. See the reference documentation for other features that can be called separately.

Get results from the service

The service returns a `200` HTTP response, and the body contains the returned data in the form of a JSON string. The following is an example of a JSON response.

```
{
  "tags":[
    {
      "name":"outdoor",
      "score":0.976
    },
    {
      "name":"bird",
      "score":0.95
    }
  ],
  "description":{
    "tags":[
      "outdoor",
      "bird"
    ],
    "captions":[
      {
        "text":"partridge in a pear tree",
        "confidence":0.96
      }
    ]
  }
}
```

See the following table for explanations of the fields in this example:

FIELD	TYPE	CONTENT
Tags	`object`	The top-level object for an array of tags.
tags[].Name	`string`	The keyword from the tags classifier.
tags[].Score	`number`	The confidence score, between 0 and 1.
description	`object`	The top-level object for an image description.
description.tags[]	`string`	The list of tags. If there is insufficient confidence in the ability to produce a caption, the tags might be the only information available to the caller.
description.captions[].text	`string`	A phrase describing the image.
description.captions[].confidence	`number`	The confidence score for the phrase.

Error codes

See the following list of possible errors and their causes:

- 400
  - InvalidImageUrl - Image URL is badly formatted or not accessible.
  - InvalidImageFormat - Input data is not a valid image.
  - InvalidImageSize - Input image is too large.
  - NotSupportedVisualFeature - Specified feature type is not valid.
  - NotSupportedImage - Unsupported image, e.g. child pornography.

    - InvalidDetails - Unsupported `detail` parameter value.
    - NotSupportedLanguage - The requested operation is not supported in the language specified.
    - BadArgument - Additional details are provided in the error message.
- 415 - Unsupported media type error. The Content-Type is not in the allowed types:
    - For an image URL: Content-Type should be application/json
    - For a binary image data: Content-Type should be application/octet-stream or multipart/form-data
- 500
    - FailedToProcess
    - Timeout - Image processing timed out.
    - InternalServerError

Next steps

To try out the REST API, go to the Image Analysis API Reference.

Analyze videos in near real time

5/12/2021 • 7 minutes to read • Edit Online

This article demonstrates how to perform near real-time analysis on frames that are taken from a live video stream by using the Computer Vision API. The basic elements of such an analysis are:

- Acquiring frames from a video source.
- Selecting which frames to analyze.
- Submitting these frames to the API.
- Consuming each analysis result that's returned from the API call.

The samples in this article are written in C#. To access the code, go to the Video frame analysis sample page on GitHub.

Approaches to running near real-time analysis

You can solve the problem of running near real-time analysis on video streams by using a variety of approaches. This article outlines three of them, in increasing levels of sophistication.

Design an infinite loop

The simplest design for near real-time analysis is an infinite loop. In each iteration of this loop, you grab a frame, analyze it, and then consume the result:

```
while (true)
{
    Frame f = GrabFrame();
    if (ShouldAnalyze(f))
    {
        AnalysisResult r = await Analyze(f);
        ConsumeResult(r);
    }
}
```

If your analysis were to consist of a lightweight, client-side algorithm, this approach would be suitable. However, when the analysis occurs in the cloud, the resulting latency means that an API call might take several seconds. During this time, you're not capturing images, and your thread is essentially doing nothing. Your maximum frame rate is limited by the latency of the API calls.

Allow the API calls to run in parallel

Although a simple, single-threaded loop makes sense for a lightweight, client-side algorithm, it doesn't fit well with the latency of a cloud API call. The solution to this problem is to allow the long-running API call to run in parallel with the frame-grabbing. In C#, you could do this by using task-based parallelism. For example, you can run the following code:

```
while (true)
{
    Frame f = GrabFrame();
    if (ShouldAnalyze(f))
    {
        var t = Task.Run(async () =>
        {
            AnalysisResult r = await Analyze(f);
            ConsumeResult(r);
        }
    }
}
```

With this approach, you launch each analysis in a separate task. The task can run in the background while you continue grabbing new frames. The approach avoids blocking the main thread as you wait for an API call to return. However, the approach can present certain disadvantages:

- It costs you some of the guarantees that the simple version provided. That is, multiple API calls might occur in parallel, and the results might get returned in the wrong order.
- It could also cause multiple threads to enter the ConsumeResult() function simultaneously, which might be dangerous if the function isn't thread-safe.
- Finally, this simple code doesn't keep track of the tasks that get created, so exceptions silently disappear. Thus, you need to add a "consumer" thread that tracks the analysis tasks, raises exceptions, kills long-running tasks, and ensures that the results get consumed in the correct order, one at a time.

Design a producer-consumer system

For your final approach, designing a "producer-consumer" system, you build a producer thread that looks similar to your previously mentioned infinite loop. However, instead of consuming the analysis results as soon as they're available, the producer simply places the tasks in a queue to keep track of them.

```
// Queue that will contain the API call tasks.
var taskQueue = new BlockingCollection<Task<ResultWrapper>>();

// Producer thread.
while (true)
{
    // Grab a frame.
    Frame f = GrabFrame();

    // Decide whether to analyze the frame.
    if (ShouldAnalyze(f))
    {
        // Start a task that will run in parallel with this thread.
        var analysisTask = Task.Run(async () =>
        {
            // Put the frame, and the result/exception into a wrapper object.
            var output = new ResultWrapper(f);
            try
            {
                output.Analysis = await Analyze(f);
            }
            catch (Exception e)
            {
                output.Exception = e;
            }
            return output;
        }

        // Push the task onto the queue.
        taskQueue.Add(analysisTask);
    }
}
```

You also create a consumer thread, which takes tasks off the queue, waits for them to finish, and either displays the result or raises the exception that was thrown. By using the queue, you can guarantee that the results get consumed one at a time, in the correct order, without limiting the maximum frame rate of the system.

```
// Consumer thread.
while (true)
{
    // Get the oldest task.
    Task<ResultWrapper> analysisTask = taskQueue.Take();

    // Wait until the task is completed.
    var output = await analysisTask;

    // Consume the exception or result.
    if (output.Exception != null)
    {
        throw output.Exception;
    }
    else
    {
        ConsumeResult(output.Analysis);
    }
}
```

Implement the solution

Get started quickly

To help get your app up and running as quickly as possible, we've implemented the system that's described in the preceding section. It's intended to be flexible enough to accommodate many scenarios, while being easy to

use. To access the code, go to the Video frame analysis sample page on GitHub.

The library contains the `FrameGrabber` class, which implements the previously discussed producer-consumer system to process video frames from a webcam. Users can specify the exact form of the API call, and the class uses events to let the calling code know when a new frame is acquired, or when a new analysis result is available.

To illustrate some of the possibilities, we've provided two sample apps that use the library.

The first sample app is a simple console app that grabs frames from the default webcam and then submits them to the Face service for face detection. A simplified version of the app is reproduced in the following code:

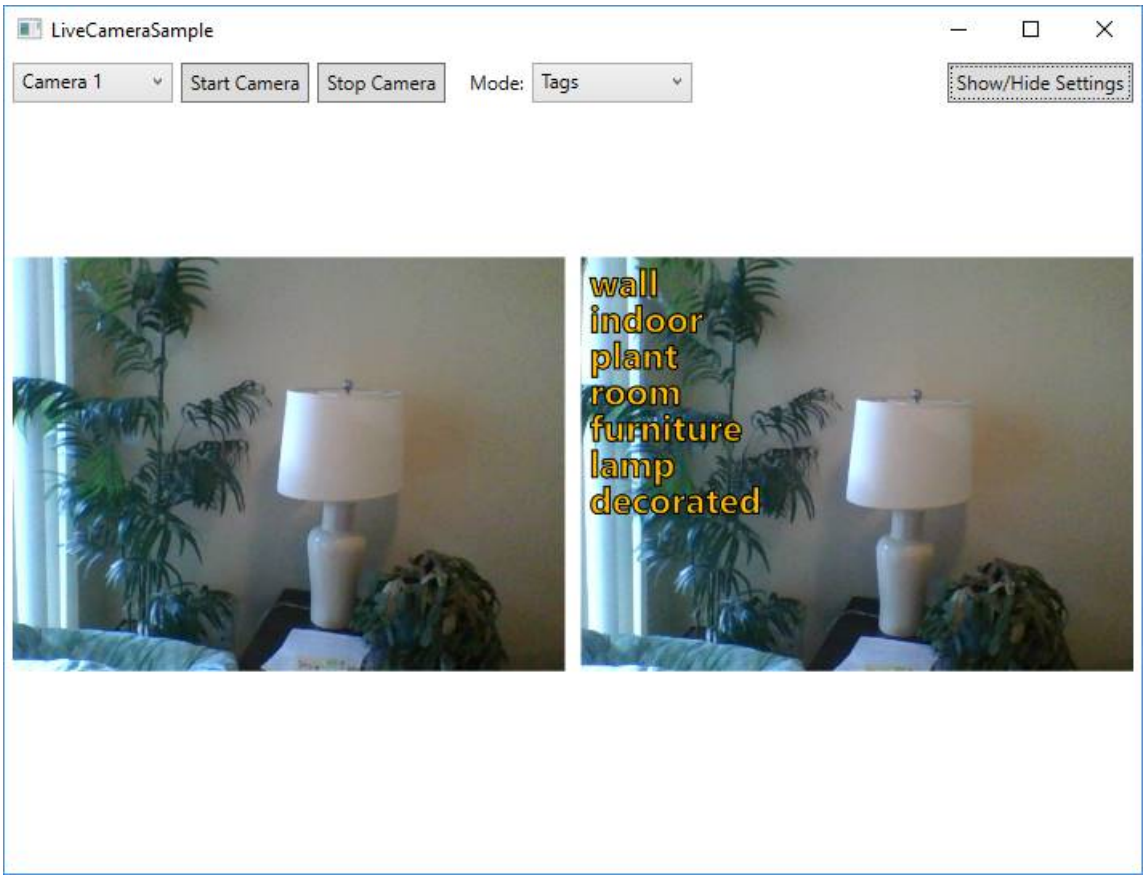
```
using System;
using System.Linq;
using Microsoft.Azure.CognitiveServices.Vision.Face;
using Microsoft.Azure.CognitiveServices.Vision.Face.Models;
using VideoFrameAnalyzer;

namespace BasicConsoleSample
{
    internal class Program
    {
        const string ApiKey = "<your API key>";
        const string Endpoint = "https://<your API region>.api.cognitive.microsoft.com";

        private static async Task Main(string[] args)
        {
            // Create grabber.
            FrameGrabber<DetectedFace[]> grabber = new FrameGrabber<DetectedFace[]>();

            // Create Face Client.
            FaceClient faceClient = new FaceClient(new ApiKeyServiceClientCredentials(ApiKey))
            {
                Endpoint = Endpoint
            };

            // Set up a listener for when we acquire a new frame.
            grabber.NewFrameProvided += (s, e) =>
            {
                Console.WriteLine($"New frame acquired at {e.Frame.Metadata.Timestamp}");
            };

            // Set up a Face API call.
            grabber.AnalysisFunction = async frame =>
            {
                Console.WriteLine($"Submitting frame acquired at {frame.Metadata.Timestamp}");
                // Encode image and submit to Face service.
                return (await
faceClient.Face.DetectWithStreamAsync(frame.Image.ToMemoryStream(".jpg"))).ToArray();
            };

            // Set up a listener for when we receive a new result from an API call.
            grabber.NewResultAvailable += (s, e) =>
            {
                if (e.TimedOut)
                    Console.WriteLine("API call timed out.");
                else if (e.Exception != null)
                    Console.WriteLine("API call threw an exception.");
                else
                    Console.WriteLine($"New result received for frame acquired at
{e.Frame.Metadata.Timestamp}. {e.Analysis.Length} faces detected");
            };

            // Tell grabber when to call the API.
            // See also TriggerAnalysisOnPredicate
            grabber.TriggerAnalysisOnInterval(TimeSpan.FromMilliseconds(3000));
```

```
            // Start running in the background.
            await grabber.StartProcessingCameraAsync();

            // Wait for key press to stop.
            Console.WriteLine("Press any key to stop...");
            Console.ReadKey();

            // Stop, blocking until done.
            await grabber.StopProcessingAsync();
        }
    }
}
```

The second sample app is a bit more interesting. It allows you to choose which API to call on the video frames. On the left side, the app shows a preview of the live video. On the right, it overlays the most recent API result on the corresponding frame.

In most modes, there's a visible delay between the live video on the left and the visualized analysis on the right. This delay is the time that it takes to make the API call. An exception is in the "EmotionsWithClientFaceDetect" mode, which performs face detection locally on the client computer by using OpenCV before it submits any images to Azure Cognitive Services.

By using this approach, you can visualize the detected face immediately. You can then update the emotions later, after the API call returns. This demonstrates the possibility of a "hybrid" approach. That is, some simple processing can be performed on the client, and then Cognitive Services APIs can be used to augment this processing with more advanced analysis when necessary.



Integrate the samples into your codebase

To get started with this sample, do the following:

1. Create an Azure account. If you already have one, you can skip to the next step.
2. Create resources for Computer Vision and Face in the Azure portal to get your key and endpoint. Make sure to select the free tier (F0) during setup.

- Computer Vision
- Face After the resources are deployed, click **Go to resource** to collect your key and endpoint for each resource.

3. Clone the Cognitive-Samples-VideoFrameAnalysis GitHub repo.
4. Open the sample in Visual Studio 2015 or later, and then build and run the sample applications:
   - For BasicConsoleSample, the Face key is hard-coded directly in BasicConsoleSample/Program.cs.
   - For LiveCameraSample, enter the keys in the **Settings** pane of the app. The keys are persisted across sessions as user data.

When you're ready to integrate the samples, reference the VideoFrameAnalyzer library from your own projects.

The image-, voice-, video-, and text-understanding capabilities of VideoFrameAnalyzer use Azure Cognitive Services. Microsoft receives the images, audio, video, and other data that you upload (via this app) and might use them for service-improvement purposes. We ask for your help in protecting the people whose data your app sends to Azure Cognitive Services.

Summary

In this article, you learned how to run near real-time analysis on live video streams by using the Face and Computer Vision services. You also learned how you can use our sample code to get started.

Feel free to provide feedback and suggestions in the GitHub repository. To provide broader API feedback, go to our UserVoice site.

Detect common objects in images

5/25/2021 • 2 minutes to read • Edit Online

Object detection is similar to tagging, but the API returns the bounding box coordinates (in pixels) for each object found. For example, if an image contains a dog, cat and person, the Detect operation will list those objects together with their coordinates in the image. You can use this functionality to process the relationships between the objects in an image. It also lets you determine whether there are multiple instances of the same tag in an image.

The Detect API applies tags based on the objects or living things identified in the image. There is currently no formal relationship between the tagging taxonomy and the object detection taxonomy. At a conceptual level, the Detect API only finds objects and living things, while the Tag API can also include contextual terms like "indoor", which can't be localized with bounding boxes.

Object detection example

The following JSON response illustrates what Computer Vision returns when detecting objects in the example image.

```
{
   "objects":[
      {
         "rectangle":{
            "x":730,
            "y":66,
            "w":135,
            "h":85
         },
         "object":"kitchen appliance",
         "confidence":0.501
      },
      {
         "rectangle":{
            "x":523,
            "y":377,
            "w":185,
            "h":46
         },
         "object":"computer keyboard",
         "confidence":0.51
      },
      {
         "rectangle":{
            "x":471,
            "y":218,
            "w":289,
            "h":226
         },
         "object":"Laptop",
         "confidence":0.85,
         "parent":{
            "object":"computer",
            "confidence":0.851
         }
      },
      {
         "rectangle":{
            "x":654,
            "y":0,
            "w":584,
            "h":473
         },
         "object":"person",
         "confidence":0.855
      }
   ],
   "requestId":"a7fde8fd-cc18-4f5f-99d3-897dcd07b308",
   "metadata":{
      "width":1260,
      "height":473,
      "format":"Jpeg"
   }
}
```


Limitations

It's important to note the limitations of object detection so you can avoid or mitigate the effects of false negatives (missed objects) and limited detail.

- Objects are generally not detected if they're small (less than 5% of the image).
- Objects are generally not detected if they're arranged closely together (a stack of plates, for example).
- Objects are not differentiated by brand or product names (different types of sodas on a store shelf, for example). However, you can get brand information from an image by using the Brand detection feature.

Use the API

The object detection feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Objects` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"objects"` section.

- Quickstart: Computer Vision REST API or client libraries

Detect popular brands in images

5/25/2021 • 2 minutes to read • Edit Online

Brand detection is a specialized mode of object detection that uses a database of thousands of global logos to identify commercial brands in images or video. You can use this feature, for example, to discover which brands are most popular on social media or most prevalent in media product placement.

The Computer Vision service detects whether there are brand logos in a given image; if so, it returns the brand name, a confidence score, and the coordinates of a bounding box around the logo.

The built-in logo database covers popular brands in consumer electronics, clothing, and more. If you find that the brand you're looking for is not detected by the Computer Vision service, you may be better served creating and training your own logo detector using the Custom Vision service.

Brand detection example

The following JSON responses illustrate what Computer Vision returns when detecting brands in the example images.


```
"brands":[
   {
      "name":"Microsoft",
      "rectangle":{
         "x":20,
         "y":97,
         "w":62,
         "h":52
      }
   }
]
```

In some cases, the brand detector will pick up both the logo image and the stylized brand name as two separate logos.


```
"brands":[
   {
      "name":"Microsoft",
      "rectangle":{
         "x":58,
         "y":106,
         "w":55,
         "h":46
      }
   },
   {
      "name":"Microsoft",
      "rectangle":{
         "x":58,
         "y":86,
         "w":202,
         "h":63
      }
   }
]
```


Use the API

The brand detection feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Brands` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"brands"` section.

- Quickstart: Computer Vision REST API or client libraries

Applying content tags to images

5/25/2021 • 2 minutes to read • Edit Online

Computer Vision returns tags based on thousands of recognizable objects, living beings, scenery, and actions. When tags are ambiguous or not common knowledge, the API response provides 'hints' to clarify the meaning of the tag in context of a known setting. Tags are not organized as a taxonomy and no inheritance hierarchies exist. A collection of content tags forms the foundation for an image 'description' displayed as human readable language formatted in complete sentences. Note, that at this point English is the only supported language for image description.

After uploading an image or specifying an image URL, Computer Vision algorithms output tags based on the objects, living beings, and actions identified in the image. Tagging is not limited to the main subject, such as a person in the foreground, but also includes the setting (indoor or outdoor), furniture, tools, plants, animals, accessories, gadgets etc.

Image tagging example

The following JSON response illustrates what Computer Vision returns when tagging visual features detected in the example image.

```
{
    "tags": [
        {
            "name": "grass",
            "confidence": 0.9999995231628418
        },
        {
            "name": "outdoor",
            "confidence": 0.99992108345031738
        },
        {
            "name": "house",
            "confidence": 0.99685388803482056
        },
        {
            "name": "sky",
            "confidence": 0.99532157182693481
        },
        {
            "name": "building",
            "confidence": 0.99436837434768677
        },
        {
            "name": "tree",
            "confidence": 0.98880356550216675
        },
        {
            "name": "lawn",
            "confidence": 0.788884699344635
        },
        {
            "name": "green",
            "confidence": 0.71250593662261963
        },
        {
            "name": "residential",
            "confidence": 0.70859086513519287
        },
        {
            "name": "grassy",
            "confidence": 0.46624681353569031
        }
    ],
    "requestId": "06f39352-e445-42dc-96fb-0a1288ad9cf1",
    "metadata": {
        "height": 200,
        "width": 300,
        "format": "Jpeg"
    }
}
```


Use the API

The tagging feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Tags` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"tags"` section.

- Quickstart: Computer Vision REST API or client libraries

Next steps

Learn the related concepts of categorizing images and describing images.

Categorize images by subject matter

5/25/2021 • 2 minutes to read • Edit Online

In addition to tags and a description, Computer Vision returns the taxonomy-based categories detected in an image. Unlike tags, categories are organized in a parent/child hereditary hierarchy, and there are fewer of them (86, as opposed to thousands of tags). All category names are in English. Categorization can be done by itself or alongside the newer tags model.

The 86-category concept

Computer vision can categorize an image broadly or specifically, using the list of 86 categories in the following diagram. For the full taxonomy in text format, see Category Taxonomy.

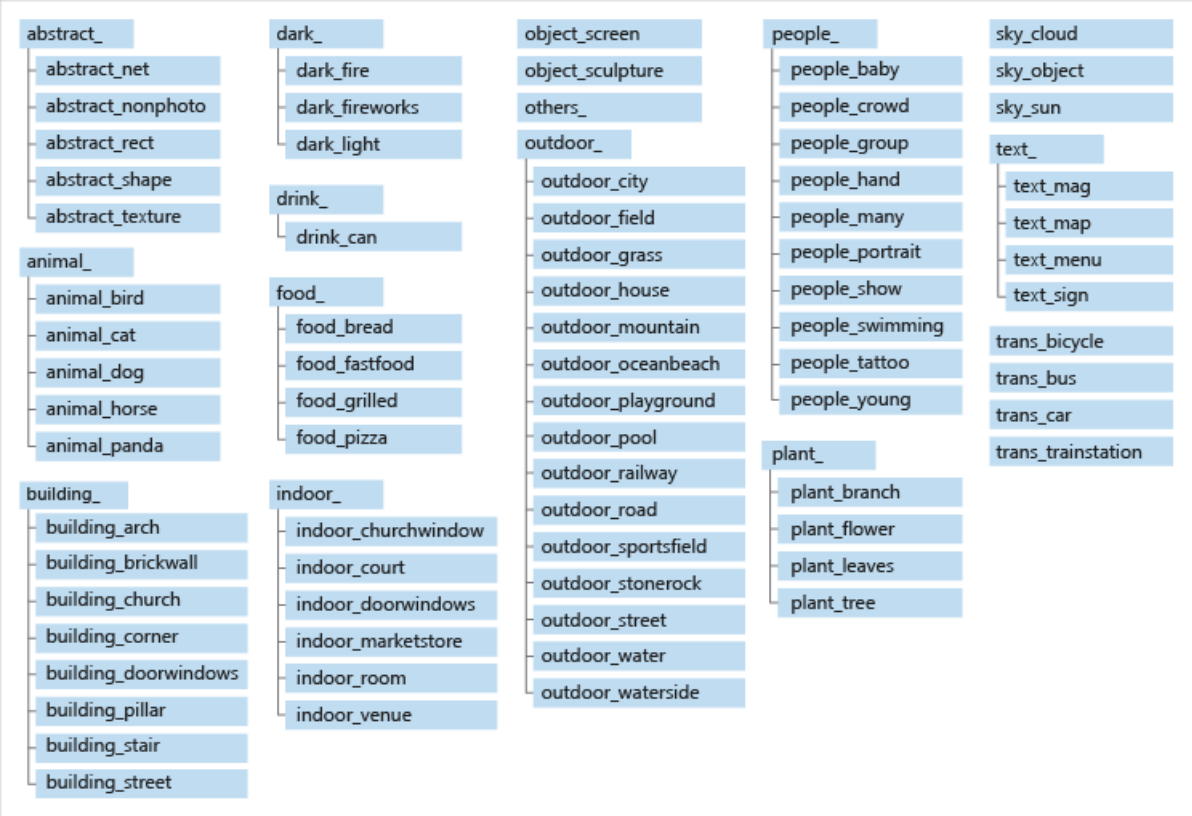


Image categorization examples

The following JSON response illustrates what Computer Vision returns when categorizing the example image based on its visual features.

```
{
    "categories": [
        {
            "name": "people_",
            "score": 0.81640625
        }
    ],
    "requestId": "bae7f76a-1cc7-4479-8d29-48a694974705",
    "metadata": {
        "height": 200,
        "width": 300,
        "format": "Jpeg"
    }
}
```


The following table illustrates a typical image set and the category returned by Computer Vision for each image.

IMAGE	CATEGORY
	people_group
	animal_dog
	outdoor_mountain

IMAGE	CATEGORY
	food_bread

Use the API

The categorization feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Categories` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"categories"` section.

- Quickstart: Computer Vision REST API or client libraries

Next steps

Learn the related concepts of tagging images and describing images.

Computer Vision 86-category taxonomy

7/26/2019 • 2 minutes to read • Edit Online

abstract_

abstract_net

abstract_nonphoto

abstract_rect

abstract_shape

abstract_texture

animal_

animal_bird

animal_cat

animal_dog

animal_horse

animal_panda

building_

building_arch

building_brickwall

building_church

building_corner

building_doorwindows

building_pillar

building_stair

building_street

dark_

drink_

drink_can

dark_fire

dark_fireworks

sky_object

food_

food_bread

food_fastfood

food_grilled

food_pizza

indoor_

indoor_churchwindow

indoor_court

indoor_doorwindows

indoor_marketstore

indoor_room

indoor_venue

dark_light

others_

outdoor_

outdoor_city

outdoor_field

outdoor_grass

outdoor_house

outdoor_mountain

outdoor_oceanbeach

outdoor_playground

outdoor_railway

outdoor_road

outdoor_sportsfield

outdoor_stonerock

outdoor_street

outdoor_water

outdoor_waterside

people_

people_baby

people_crowd

people_group

people_hand

people_many

people_portrait

people_show

people_tattoo

people_young

plant_

plant_branch

plant_flower

plant_leaves

plant_tree

object_screen

object_sculpture

sky_cloud

sky_sun

people_swimming

outdoor_pool

text_

text_mag

text_map

text_menu

text_sign

trans_bicycle

trans_bus

trans_car

trans_trainstation

Describe images with human-readable language

5/25/2021 • 2 minutes to read • Edit Online

Computer Vision can analyze an image and generate a human-readable sentence that describes its contents. The algorithm actually returns several descriptions based on different visual features, and each description is given a confidence score. The final output is a list of descriptions ordered from highest to lowest confidence.

Image description example

The following JSON response illustrates what Computer Vision returns when describing the example image based on its visual features.


```
{
    "description": {
        "tags": ["outdoor", "building", "photo", "city", "white", "black", "large", "sitting", "old",
"water", "skyscraper", "many", "boat", "river", "group", "street", "people", "field", "tall", "bird",
"standing"],
        "captions": [
            {
                "text": "a black and white photo of a city",
                "confidence": 0.95301952483304808
            },
            {
                "text": "a black and white photo of a large city",
                "confidence": 0.94085190563213816
            },
            {
                "text": "a large white building in a city",
                "confidence": 0.93108362931954824
            }
        ]
    },
    "requestId": "b20bfc83-fb25-4b8d-a3f8-b2a1f084b159",
    "metadata": {
        "height": 300,
        "width": 239,
        "format": "Jpeg"
    }
}
```


Use the API

The image description feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Description` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"description"` section.

- Quickstart: Computer Vision REST API or client libraries

Next steps

Learn the related concepts of tagging images and categorizing images.

Face detection with Computer Vision

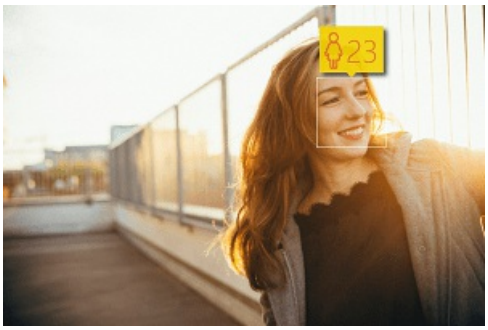
5/25/2021 • 2 minutes to read • Edit Online

Computer Vision can detect human faces within an image and generate the age, gender, and rectangle for each detected face.

> **NOTE**
> This feature is also offered by the Azure Face service. See this alternative for more detailed face analysis, including face identification and pose detection.

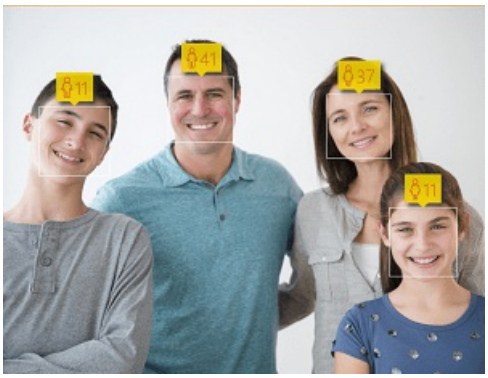
Face detection examples

The following example demonstrates the JSON response returned by Computer Vision for an image containing a single human face.




```
{
    "faces": [
        {
            "age": 23,
            "gender": "Female",
            "faceRectangle": {
                "top": 45,
                "left": 194,
                "width": 44,
                "height": 44
            }
        }
    ],
    "requestId": "8439ba87-de65-441b-a0f1-c85913157ecd",
    "metadata": {
        "height": 200,
        "width": 300,
        "format": "Png"
    }
}
```


The next example demonstrates the JSON response returned for an image containing multiple human faces.




```
{
    "faces": [
        {
            "age": 11,
            "gender": "Male",
            "faceRectangle": {
                "top": 62,
                "left": 22,
                "width": 45,
                "height": 45
            }
        },
        {
            "age": 11,
            "gender": "Female",
            "faceRectangle": {
                "top": 127,
                "left": 240,
                "width": 42,
                "height": 42
            }
        },
        {
            "age": 37,
            "gender": "Female",
            "faceRectangle": {
                "top": 55,
                "left": 200,
                "width": 41,
                "height": 41
            }
        },
        {
            "age": 41,
            "gender": "Male",
            "faceRectangle": {
                "top": 45,
                "left": 103,
                "width": 39,
                "height": 39
            }
        }
    ],
    "requestId": "3a383cbe-1a05-4104-9ce7-1b5cf352b239",
    "metadata": {
        "height": 230,
        "width": 300,
        "format": "Png"
    }
}
```


Use the API

The face detection feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Faces` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"faces"` section.

- Quickstart: Computer Vision REST API or client libraries

Detecting image types with Computer Vision

5/25/2021 • 2 minutes to read • Edit Online

With the Analyze Image API, Computer Vision can analyze the content type of images, indicating whether an image is clip art or a line drawing.

Detecting clip art

Computer Vision analyzes an image and rates the likelihood of the image being clip art on a scale of 0 to 3, as described in the following table.

VALUE	MEANING
0	Non-clip-art
1	Ambiguous
2	Normal-clip-art
3	Good-clip-art

Clip art detection examples

The following JSON responses illustrates what Computer Vision returns when rating the likelihood of the example images being clip art.

```
{
    "imageType": {
        "clipArtType": 3,
        "lineDrawingType": 0
    },
    "requestId": "88c48d8c-80f3-449f-878f-6947f3b35a27",
    "metadata": {
        "height": 225,
        "width": 300,
        "format": "Jpeg"
    }
}
```

```
{
    "imageType": {
        "clipArtType": 0,
        "lineDrawingType": 0
    },
    "requestId": "a9c8490a-2740-4e04-923b-e8f4830d0e47",
    "metadata": {
        "height": 200,
        "width": 300,
        "format": "Jpeg"
    }
}
```

Detecting line drawings

Computer Vision analyzes an image and returns a boolean value indicating whether the image is a line drawing.

Line drawing detection examples

The following JSON responses illustrates what Computer Vision returns when indicating whether the example images are line drawings.

```
{
    "imageType": {
        "clipArtType": 2,
        "lineDrawingType": 1
    },
    "requestId": "6442dc22-476a-41c4-aa3d-9ceb15172f01",
    "metadata": {
        "height": 268,
        "width": 300,
        "format": "Jpeg"
    }
}
```


```
{
    "imageType": {
        "clipArtType": 0,
        "lineDrawingType": 0
    },
    "requestId": "98437d65-1b05-4ab7-b439-7098b5dfdcbf",
    "metadata": {
        "height": 200,
        "width": 300,
        "format": "Jpeg"
    }
}
```


Use the API

The image type detection feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `ImageType` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"imageType"` section.

- Quickstart: Computer Vision REST API or client libraries

Detect domain-specific content

5/25/2021 • 2 minutes to read • Edit Online

In addition to tagging and high-level categorization, Computer Vision also supports further domain-specific analysis using models that have been trained on specialized data.

There are two ways to use the domain-specific models: by themselves (scoped analysis) or as an enhancement to the categorization feature.

Scoped analysis

You can analyze an image using only the chosen domain-specific model by calling the Models/<model>/Analyze API.

The following is a sample JSON response returned by the **models/celebrities/analyze** API for the given image:

```
{
  "result": {
    "celebrities": [{
      "faceRectangle": {
        "top": 391,
        "left": 318,
        "width": 184,
        "height": 184
      },
      "name": "Satya Nadella",
      "confidence": 0.99999856948852539
    }]
  },
  "requestId": "8217262a-1a90-4498-a242-68376a4b956b",
  "metadata": {
    "width": 800,
    "height": 1200,
    "format": "Jpeg"
  }
}
```


Enhanced categorization analysis

You can also use domain-specific models to supplement general image analysis. You do this as part of high-level categorization by specifying domain-specific models in the *details* parameter of the Analyze API call.

In this case, the 86-category taxonomy classifier is called first. If any of the detected categories have a matching domain-specific model, the image is passed through that model as well and the results are added.

The following JSON response shows how domain-specific analysis can be included as the `detail` node in a broader categorization analysis.


```
"categories":[
  {
    "name":"abstract_",
    "score":0.00390625
  },
  {
    "name":"people_",
    "score":0.83984375,
    "detail":{
      "celebrities":[
        {
          "name":"Satya Nadella",
          "faceRectangle":{
            "left":597,
            "top":162,
            "width":248,
            "height":248
          },
          "confidence":0.999028444
        }
      ],
      "landmarks":[
        {
          "name":"Forbidden City",
          "confidence":0.9978346
        }
      ]
    }
  }
]
```

List the domain-specific models

Currently, Computer Vision supports the following domain-specific models:

NAME	DESCRIPTION
celebrities	Celebrity recognition, supported for images classified in the `people_` category
landmarks	Landmark recognition, supported for images classified in the `outdoor_` or `building_` categories

Calling the Models API will return this information along with the categories to which each model can apply:

```
{
  "models":[
    {
      "name":"celebrities",
      "categories":[
        "people_",
        "人_",
        "pessoas_",
        "gente_"
      ]
    },
    {
      "name":"landmarks",
      "categories":[
        "outdoor_",
        "户外_",
        "屋外_",
        "aoarlivre_",
        "alairelibre_",
        "building_",
        "建筑_",
        "建物_",
        "edifício_"
      ]
    }
  ]
}
```

Use the API

This feature is available through the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Celebrities` or `Landmarks` in the **details** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"details"` section.

- Quickstart: Computer Vision REST API or client libraries

Detect color schemes in images

5/25/2021 • 2 minutes to read • Edit Online

Computer Vision analyzes the colors in an image to provide three different attributes: the dominant foreground color, the dominant background color, and the set of dominant colors for the image as a whole. Returned colors belong to the set: black, blue, brown, gray, green, orange, pink, purple, red, teal, white, and yellow.

Computer Vision also extracts an accent color, which represents the most vibrant color in the image, based on a combination of dominant colors and saturation. The accent color is returned as a hexadecimal HTML color code.

Computer Vision also returns a boolean value indicating whether an image is in black and white.

Color scheme detection examples

The following example illustrates the JSON response returned by Computer Vision when detecting the color scheme of the example image. In this case, the example image is not a black and white image, but the dominant foreground and background colors are black, and the dominant colors for the image as a whole are black and white.


```
{
    "color": {
        "dominantColorForeground": "Black",
        "dominantColorBackground": "Black",
        "dominantColors": ["Black", "White"],
        "accentColor": "BB6D10",
        "isBwImg": false
    },
    "requestId": "0dc394bf-db50-4871-bdcc-13707d9405ea",
    "metadata": {
        "height": 202,
        "width": 300,
        "format": "Jpeg"
    }
}
```


Dominant color examples

The following table shows the returned foreground, background, and image colors for each sample image.

IMAGE	DOMINANT COLORS
	Foreground: Black Background: White Colors: Black, White, Green
	Foreground: Black Background: Black Colors: Black

Accent color examples

The following table shows the returned accent color, as a hexadecimal HTML color value, for each example image.

IMAGE	ACCENT COLOR
	#BB6D10
	#C6A205

IMAGE	ACCENT COLOR
	#474A84

Black & white detection examples

The following table shows Computer Vision's black and white evaluation in the sample images.

IMAGE	BLACK & WHITE?
	true
	false

Use the API

The color scheme detection feature is part of the Analyze Image API. You can call this API through a native SDK or through REST calls. Include `Color` in the **visualFeatures** query parameter. Then, when you get the full JSON response, simply parse the string for the contents of the `"color"` section.

- Quickstart: Computer Vision REST API or client libraries

Generating smart-cropped thumbnails with Computer Vision

5/25/2021 • 2 minutes to read • Edit Online

A thumbnail is a reduced-size representation of an image. Thumbnails are used to represent images and other data in a more economical, layout-friendly way. The Computer Vision API uses smart cropping, together with resizing the image, to create intuitive thumbnails for a given image.

The Computer Vision thumbnail generation algorithm works as follows:

1. Remove distracting elements from the image and identify the *area of interest*—the area of the image in which the main object(s) appears.
2. Crop the image based on the identified *area of interest*.
3. Change the aspect ratio to fit the target thumbnail dimensions.

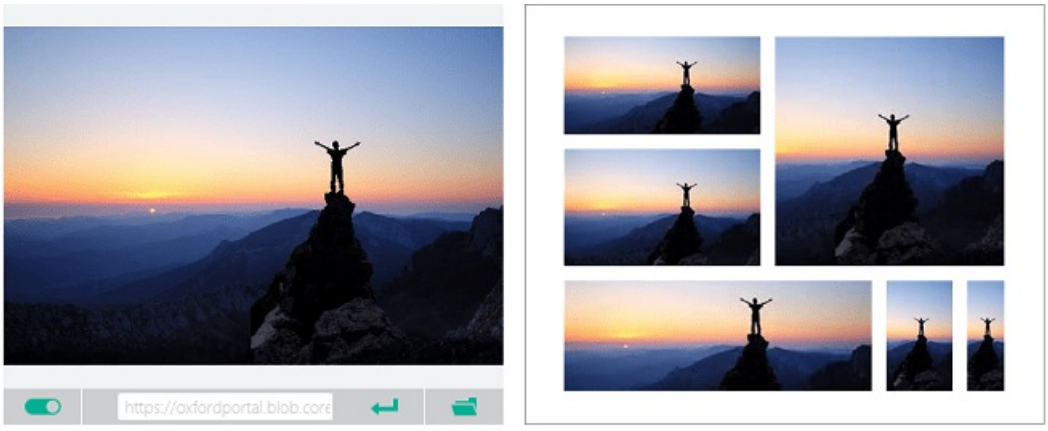
Area of interest

When you upload an image, the Computer Vision API analyzes it to determine the *area of interest*. It can then use this region to determine how to crop the image. The cropping operation, however, will always match the desired aspect ratio if one is specified.

You can also get the raw bounding box coordinates of this same *area of interest* by calling the **areaOfInterest** API instead. You can then use this information to modify the original image however you wish.

Examples

The generated thumbnail can vary widely depending on what you specify for height, width, and smart cropping, as shown in the following image.



The following table illustrates typical thumbnails generated by Computer Vision for the example images. The thumbnails were generated for a specified target height and width of 50 pixels, with smart cropping enabled.

IMAGE	THUMBNAIL
	
	
	

Use the API

The generate thumbnail feature is available through the Get Thumbnail and Get Area of Interest APIs. You can call this API through a native SDK or through REST calls.

- Quickstart: Computer Vision REST API or client libraries

Detect adult content

5/25/2021 • 2 minutes to read • Edit Online

Computer Vision can detect adult material in images so that developers can restrict the display of these images in their software. Content flags are applied with a score between zero and one so developers can interpret the results according to their own preferences.

> **NOTE**
>
> Much of this functionality is offered by the Azure Content Moderator service. See this alternative for solutions to more rigorous content moderation scenarios, such as text moderation and human review workflows.

Content flag definitions

The "adult" classification contains several different categories:

- **Adult** images are explicitly sexual in nature and often show nudity and sexual acts.
- **Racy** images are sexually suggestive in nature and often contain less sexually explicit content than images tagged as **Adult**.
- **Gory** images show blood/gore.

Use the API

You can detect adult content with the Analyze Image API. When you add the value of `Adult` to the **visualFeatures** query parameter, the API returns three boolean properties— `isAdultContent`, `isRacyContent`, and `isGoryContent`—in its JSON response. The method also returns corresponding properties— `adultScore`, `racyScore`, and `goreScore`—which represent confidence scores between zero and one for each respective category.

- Quickstart: Computer Vision REST API or client libraries

Tutorial: Use Computer Vision to generate image metadata in Azure Storage

3/5/2021 • 6 minutes to read • Edit Online

In this tutorial, you'll learn how to integrate the Azure Computer Vision service into a web app to generate metadata for uploaded images. This is useful for digital asset management (DAM) scenarios, such as if a company wants to quickly generate descriptive captions or searchable keywords for all of its images.

A full app guide can be found in the Azure Storage and Cognitive Services Lab on GitHub, and this tutorial essentially covers Exercise 5 of the lab. You may want to create the full application by following every step, but if you only want to learn how to integrate Computer Vision into an existing web app, read along here.

This tutorial shows you how to:

- Create a Computer Vision resource in Azure
- Perform image analysis on Azure Storage images
- Attach metadata to Azure Storage images
- Check image metadata using Azure Storage Explorer

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

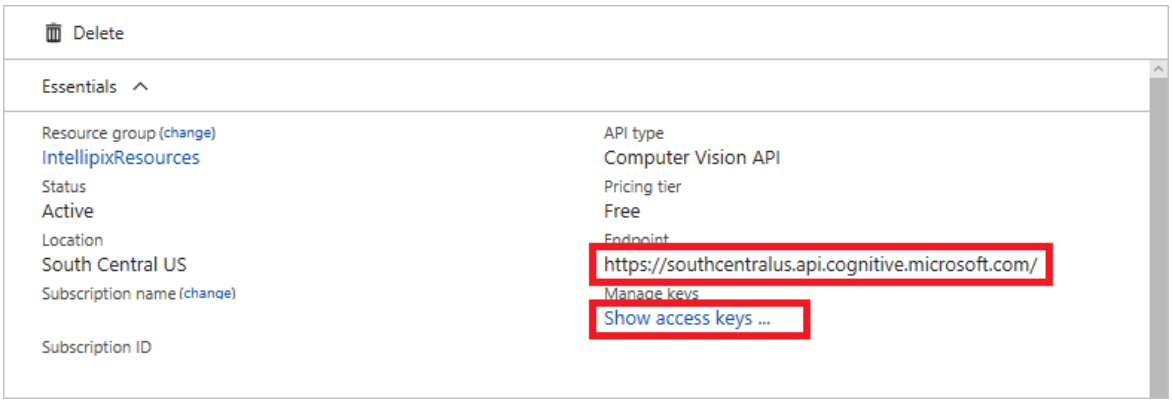
- Visual Studio 2017 Community edition or higher, with the "ASP.NET and web development" and "Azure development" workloads installed.
- An Azure Storage account with a blob container set up for image storage (follow Exercises 1 of the Azure Storage Lab if you need help with this step).
- The Azure Storage Explorer tool (follow Exercise 2 of the Azure Storage Lab if you need help with this step).
- An ASP.NET web application with access to Azure Storage (follow Exercise 3 of the Azure Storage Lab to create such an app quickly).

Create a Computer Vision resource

You'll need to create a Computer Vision resource for your Azure account; this resource manages your access to Azure's Computer Vision service.

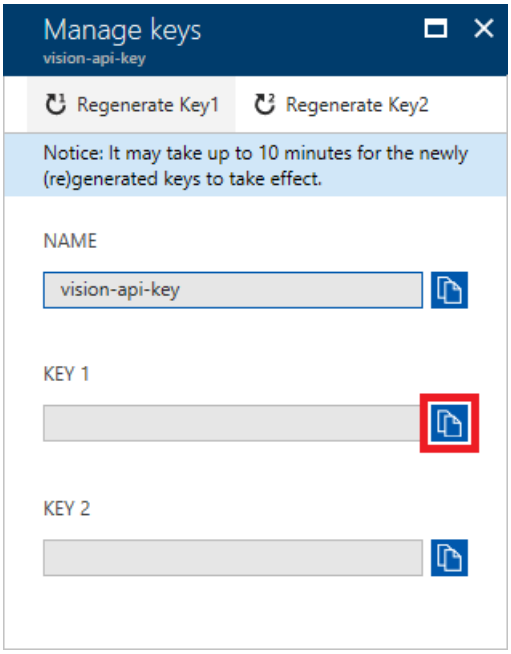
1. Follow the instructions in Create an Azure Cognitive Services resource to create a Computer Vision resource.

2. Then go to the menu for your resource group and click the Computer Vision API subscription that you just created. Copy the URL under **Endpoint** to somewhere you can easily retrieve it in a moment. Then click **Show access keys**.



> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

3. In the next window, copy the value of **KEY 1** to the clipboard.



Add Computer Vision credentials

Next, you'll add the required credentials to your app so that it can access Computer Vision resources.

Open your ASP.NET web application in Visual Studio and navigate to the **Web.config** file at the root of the project. Add the following statements to the `<appSettings>` section of the file, replacing `VISION_KEY` with the key you copied in the previous step, and `VISION_ENDPOINT` with the URL you saved in the step before.

```
<add key="SubscriptionKey" value="VISION_KEY" />
<add key="VisionEndpoint" value="VISION_ENDPOINT" />
```

Then in the Solution Explorer, right-click the project and use the **Manage NuGet Packages** command to install the package **Microsoft.Azure.CognitiveServices.Vision.ComputerVision**. This package contains the types needed to call the Computer Vision API.

Add metadata generation code

Next, you'll add the code that actually leverages the Computer Vision service to create metadata for images. These steps will apply to the ASP.NET app in the lab, but you can adapt them to your own app. What's important is that at this point you have an ASP.NET web application that can upload images to an Azure Storage container, read images from it, and display them in the view. If you're unsure about this step, it's best to follow Exercise 3 of the Azure Storage Lab.

1. Open the *HomeController.cs* file in the project's **Controllers** folder and add the following `using` statements at the top of the file:

```
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
```

2. Then, go to the **Upload** method; this method converts and uploads images to blob storage. Add the following code immediately after the block that begins with `// Generate a thumbnail` (or at the end of your image-blob-creation process). This code takes the blob containing the image (`photo`), and uses Computer Vision to generate a description for that image. The Computer Vision API also generates a list of keywords that apply to the image. The generated description and keywords are stored in the blob's metadata so that they can be retrieved later on.

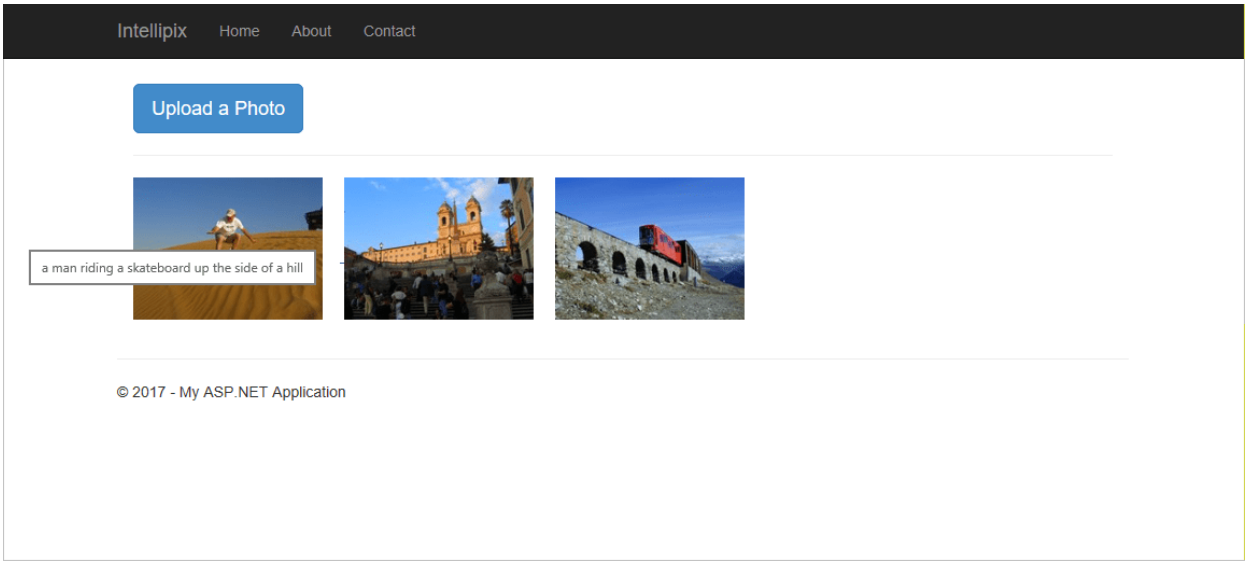
```
// Submit the image to Azure's Computer Vision API
ComputerVisionClient vision = new ComputerVisionClient(
    new ApiKeyServiceClientCredentials(ConfigurationManager.AppSettings["SubscriptionKey"]),
    new System.Net.Http.DelegatingHandler[] { });
vision.Endpoint = ConfigurationManager.AppSettings["VisionEndpoint"];

VisualFeatureTypes[] features = new VisualFeatureTypes[] { VisualFeatureTypes.Description };
var result = await vision.AnalyzeImageAsync(photo.Uri.ToString(), features);

// Record the image description and tags in blob metadata
photo.Metadata.Add("Caption", result.Description.Captions[0].Text);

for (int i = 0; i < result.Description.Tags.Count; i++)
{
    string key = String.Format("Tag{0}", i);
    photo.Metadata.Add(key, result.Description.Tags[i]);
}

await photo.SetMetadataAsync();
```

3. Next, go to the **Index** method in the same file. This method enumerates the stored image blobs in the targeted blob container (as **IListBlobItem** instances) and passes them to the application view. Replace the `foreach` block in this method with the following code. This code calls **CloudBlockBlob.FetchAttributes** to get each blob's attached metadata. It extracts the computer-generated description (`caption`) from the metadata and adds it to the **BlobInfo** object, which gets passed to the view.

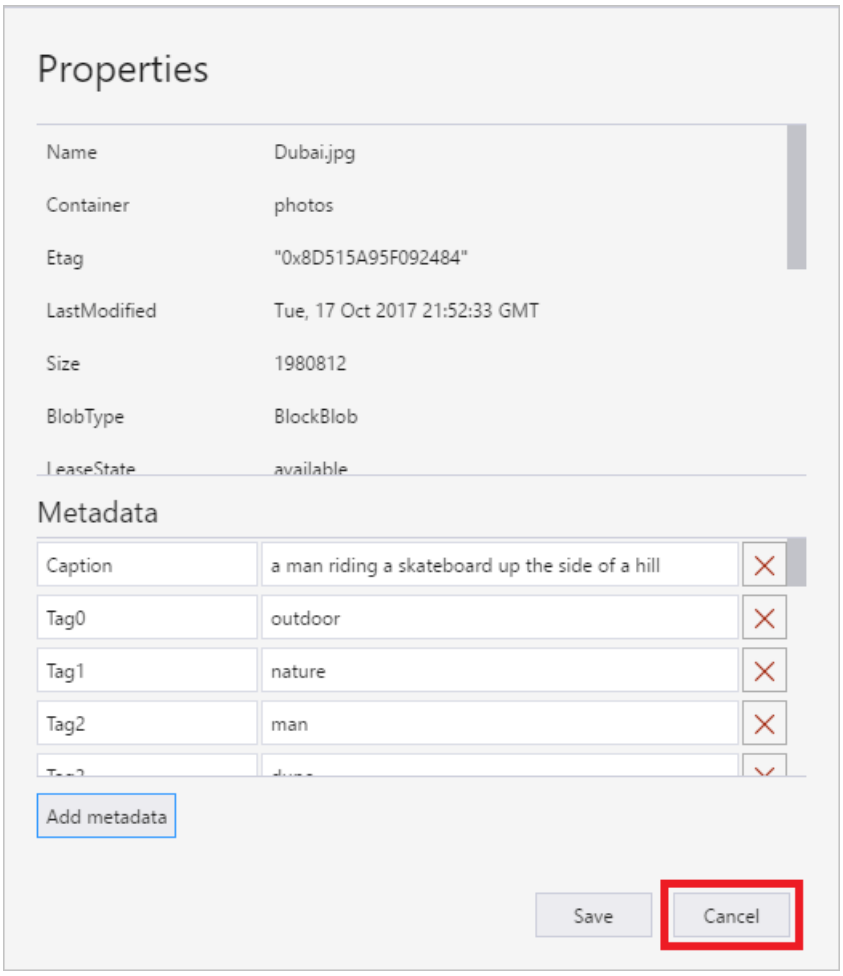
```
foreach (IListBlobItem item in container.ListBlobs())
{
    var blob = item as CloudBlockBlob;

    if (blob != null)
    {
        blob.FetchAttributes(); // Get blob metadata
        var caption = blob.Metadata.ContainsKey("Caption") ? blob.Metadata["Caption"] : blob.Name;

        blobs.Add(new BlobInfo()
        {
            ImageUri = blob.Uri.ToString(),
            ThumbnailUri = blob.Uri.ToString().Replace("/photos/", "/thumbnails/"),
            Caption = caption
        });
    }
}
```

Test the app

Save your changes in Visual Studio and press **Ctrl+F5** to launch the application in your browser. Use the app to upload a few images, either from the "photos" folder in the lab's resources or from your own folder. When you hover the cursor over one of the images in the view, a tooltip window should appear and display the computer-generated caption for the image.



To view all of the attached metadata, use the Azure Storage Explorer to view the storage container you're using for images. Right-click any of the blobs in the container and select **Properties**. In the dialog, you'll see a list of key-value pairs. The computer-generated image description is stored in the item "Caption," and the search keywords are stored in "Tag0," "Tag1," and so on. When you're finished, click **Cancel** to close the dialog.



Clean up resources

If you'd like to keep working on your web app, see the Next steps section. If you don't plan to continue using this application, you should delete all app-specific resources. To do delete resources, you can delete the resource group that contains your Azure Storage subscription and Computer Vision resource. This will remove the storage account, the blobs uploaded to it, and the App Service resource needed to connect with the ASP.NET web app.

To delete the resource group, open the **Resource groups** tab in the portal, navigate to the resource group you used for this project, and click **Delete resource group** at the top of the view. You'll be asked to type the resource group's name to confirm you want to delete it, because once deleted, a resource group can't be recovered.

Next steps

In this tutorial, you set up Azure's Computer Vision service in an existing web app to automatically generate captions and keywords for blob images as they're uploaded. Next, refer to the Azure Storage Lab, Exercise 6, to learn how to add search functionality to your web app. This takes advantage of the search keywords that the Computer Vision service generates.

Add search to your app

What is Spatial Analysis?

6/22/2021 • 2 minutes to read • Edit Online

The Spatial Analysis service helps organizations maximize the value of their physical spaces by understanding people's movements and presence within a given area. It allows you to ingest video from CCTV or surveillance cameras, run AI operations to extract insights from the video streams, and generate events to be used by other systems. With input from a camera stream, an AI operation can do things like count the number of people entering a space or measure compliance with face mask and social distancing guidelines.

What it does

The core operations of Spatial Analysis are all built on a pipeline that ingests video, detects people in the video, tracks the people as they move around over time, and generates events as people interact with regions of interest.

Spatial Analysis features

FEATURE	DEFINITION
People Detection	This component answers the question, "Where are the people in this image?" It finds people in an image and passes a bounding box indicating the location of each person to the people tracking component.
People Tracking	This component connects the people detections over time as people move around in front of a camera. It uses temporal logic about how people typically move and basic information about the overall appearance of the people. It does not track people across multiple cameras. If a person exits the field of view for longer than approximately a minute and then re-enters the camera view, the system will perceive this as a new person. People Tracking does not uniquely identify individuals across cameras. It does not use facial recognition or gait tracking.
Face Mask Detection	This component detects the location of a person's face in the camera's field of view and identifies the presence of a face mask. The AI operation scans images from video; where a face is detected the service provides a bounding box around the face. Using object detection capabilities, it identifies the presence of face masks within the bounding box. Face Mask detection does not involve distinguishing one face from another face, predicting or classifying facial attributes or performing facial recognition.
Region of Interest	This is a user-defined zone or line in the input video frame. When a person interacts with this region on the video, the system generates an event. For example, for the PersonCrossingLine operation, a line is defined in the video. When a person crosses that line an event is generated.

FEATURE	DEFINITION
Event	An event is the primary output of Spatial Analysis. Each operation emits a specific event either periodically (like once per minute) or whenever a specific trigger occurs. The event includes information about what occurred in the input video but does not include any images or video. For example, the PeopleCount operation can emit an event containing the updated count every time the count of people changes (trigger) or once every minute (periodically).

Get started

Follow the quickstart to set up the container and begin analyzing video.

Responsible use of Spatial Analysis technology

To learn how to use Spatial Analysis technology responsibly, see the transparency note. Microsoft's transparency notes are intended to help you understand how our AI technology works, the choices system owners can make that influence system performance and behavior, and the importance of thinking about the whole system, including the technology, the people, and the environment.

Next steps

Quickstart: Spatial Analysis container

Install and run the Spatial Analysis container (Preview)

6/28/2021 • 18 minutes to read • Edit Online

The Spatial Analysis container enables you to analyze real-time streaming video to understand spatial relationships between people, their movement, and interactions with objects in physical environments. Containers are great for specific security and data governance requirements.

Prerequisites

- Azure subscription - Create one for free
- Once you have your Azure subscription, create a Computer Vision resource for the Standard S1 tier in the Azure portal to get your key and endpoint. After it deploys, click **Go to resource**.
  - You will need the key and endpoint from the resource you create to run the Spatial Analysis container. You'll use your key and endpoint later.

Spatial Analysis container requirements

To run the Spatial Analysis container, you need a compute device with a NVIDIA Tesla T4 GPU. We recommend that you use Azure Stack Edge with GPU acceleration, however the container runs on any other desktop machine that meets the minimum requirements. We will refer to this device as the host computer.

- Azure Stack Edge device
- Desktop machine
- Azure VM with GPU

Azure Stack Edge is a Hardware-as-a-Service solution and an AI-enabled edge computing device with network data transfer capabilities. For detailed preparation and setup instructions, see the Azure Stack Edge documentation.

REQUIREMENT	DESCRIPTION
Camera	The Spatial Analysis container is not tied to a specific camera brand. The camera device needs to: support Real-Time Streaming Protocol(RTSP) and H.264 encoding, be accessible to the host computer, and be capable of streaming at 15FPS and 1080p resolution.
Linux OS	Ubuntu Desktop 18.04 LTS must be installed on the host computer.

Set up the host computer

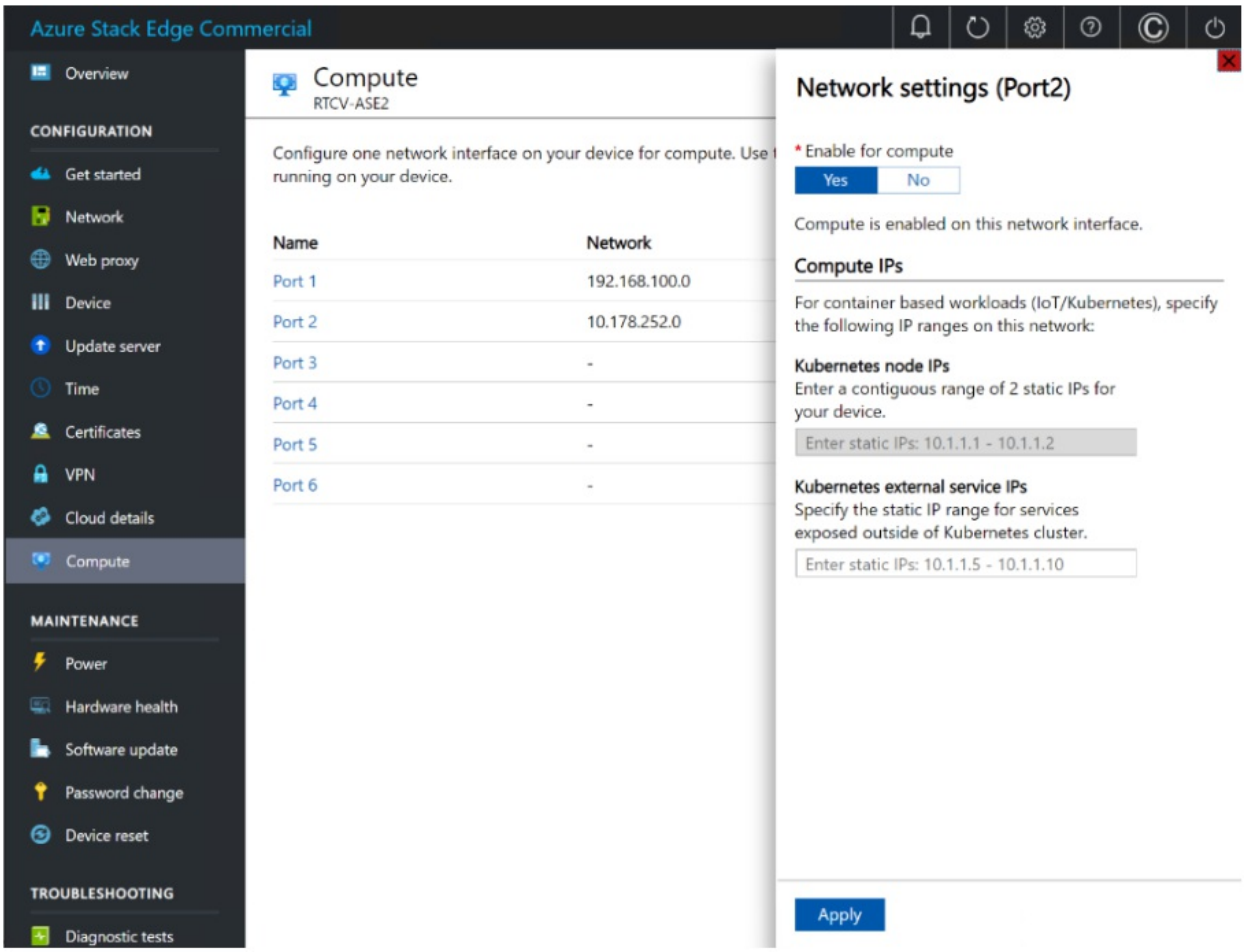
It is recommended that you use an Azure Stack Edge device for your host computer. Click **Desktop Machine** if you're configuring a different device, or **Virtual Machine** if you're utilizing a VM.

- Azure Stack Edge device
- Desktop machine
- Azure VM with GPU

Configure compute on the Azure Stack Edge portal

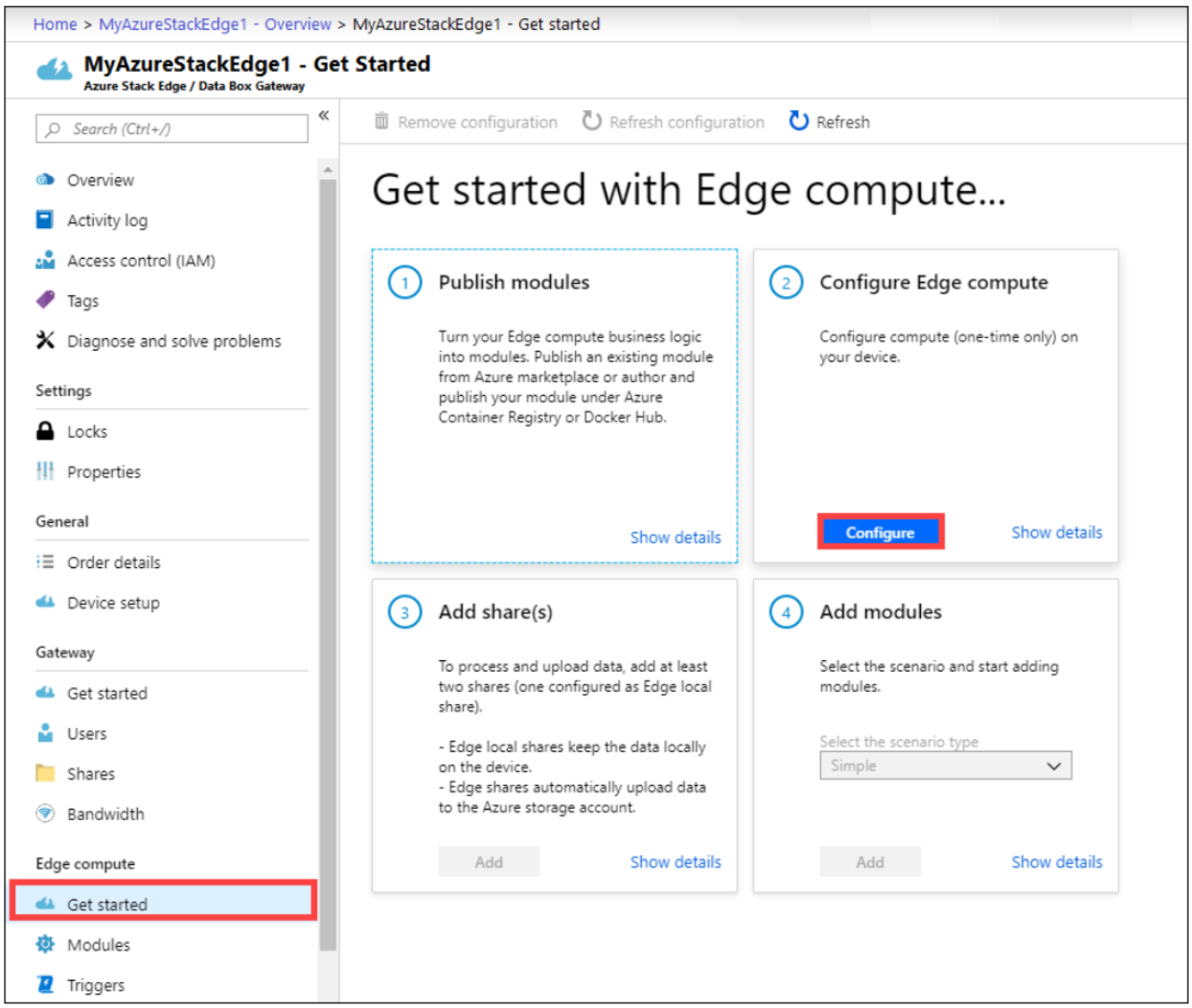
Spatial Analysis uses the compute features of the Azure Stack Edge to run an AI solution. To enable the compute features, make sure that:

- You've connected and activated your Azure Stack Edge device.
- You have a Windows client system running PowerShell 5.0 or later, to access the device.
- To deploy a Kubernetes cluster, you need to configure your Azure Stack Edge device via the **Local UI** on the Azure portal:
  1. Enable the compute feature on your Azure Stack Edge device. To enable compute, go to the **Compute** page in the web interface for your device.
  2. Select a network interface that you want to enable for compute, then click **Enable**. This will create a virtual switch on your device, on that network interface.
  3. Leave the Kubernetes test node IP addresses and the Kubernetes external services IP addresses blank.
  4. Click **Apply**. This operation may take about two minutes.



Set up an Edge compute role and create an IoT Hub resource

In the Azure portal, navigate to your Azure Stack Edge resource. On the **Overview** page or navigation list, click the Edge compute **Get started** button. In the**Configure Edge compute**tile, click **Configure**.



In the**Configure Edge compute**page, choose an existing IoT Hub, or choose to create a new one. By default, a Standard (S1) pricing tier is used to create an IoT Hub resource. To use a free tier IoT Hub resource, create one and then select it. The IoT Hub resource uses the same subscription and resource group that is used by the Azure Stack Edge resource

Click **Create**. The IoT Hub resource creation may take a couple of minutes. After the IoT Hub resource is created, the**Configure Edge compute** tile will update to show the new configuration. To confirm that the Edge compute role has been configured, select**View config** on the**Configure compute**tile.

When the Edge compute role is set up on the Edge device, it creates two devices: an IoT device and an IoT Edge device. Both devices can be viewed in the IoT Hub resource. The Azure IoT Edge Runtime will already be running on the IoT Edge device.

> **NOTE**
>
> - Currently only the Linux platform is supported for IoT Edge devices. For help troubleshooting the Azure Stack Edge device, see the logging and troubleshooting article.
> - To learn more about how to configure an IoT Edge device to communicate through a proxy server, see Configure an IoT Edge device to communicate through a proxy server

Enable MPS on Azure Stack Edge

1. Run a Windows PowerShell session as an Administrator.
2. Make sure that the Windows Remote Management service is running on your client. In the PowerShell terminal, use the following command

```
winrm quickconfig
```

If you see warnings about a firewall exception, check your network connection type, and see the Windows Remote Management documentation.

3. Assign a variable to the device IP address.

```
$ip = "<device-IP-address>"
```

4. To add the IP address of your device to the client's trusted hosts list, use the following command:

```
Set-Item WSMan:\localhost\Client\TrustedHosts $ip -Concatenate -Force
```

5. Start a Windows PowerShell session on the device.

```
Enter-PSSession -ComputerName $ip -Credential $ip\EdgeUser -ConfigurationName Minishell
```

6. Provide the password when prompted. Use the same password that is used to sign into the local web UI. The default local web UI password is `Password1`.

Type `Start-HcsGpuMPS` to start the MPS service on the device.

For help troubleshooting the Azure Stack Edge device, see Troubleshooting the Azure Stack Edge device

IoT Deployment manifest

To streamline container deployment on multiple host computers, you can create a deployment manifest file to specify the container creation options, and environment variables. You can find an example of a deployment manifest for Azure Stack Edge, other desktop machines, and Azure VM with GPU on GitHub.

The following table shows the various Environment Variables used by the IoT Edge Module. You can also set them in the deployment manifest linked above, using the `env` attribute in `spatialanalysis`:

SETTING NAME	VALUE	DESCRIPTION
ARCHON_LOG_LEVEL	Info; Verbose	Logging level, select one of the two values
ARCHON_SHARED_BUFFER_LIMIT	377487360	Do not modify
ARCHON_PERF_MARKER	false	Set this to true for performance logging, otherwise this should be false
ARCHON_NODES_LOG_LEVEL	Info; Verbose	Logging level, select one of the two values
OMP_WAIT_POLICY	PASSIVE	Do not modify
QT_X11_NO_MITSHM	1	Do not modify
APIKEY	your API Key	Collect this value from Azure portal from your Computer Vision resource. You can find it in the **Key and endpoint** section for your resource.

SETTING NAME	VALUE	DESCRIPTION
BILLING	your Endpoint URI	Collect this value from Azure portal from your Computer Vision resource. You can find it in the **Key and endpoint** section for your resource.
EULA	accept	This value needs to be set to *accept* for the container to run
DISPLAY	:1	This value needs to be same as the output of `echo $DISPLAY` on the host computer. Azure Stack Edge devices do not have a display. This setting is not applicable
ARCHON_GRAPH_READY_TIMEOUT	600	Add this environment variable if your GPU is **not** T4 or NVIDIA 2080 Ti
ORT_TENSORRT_ENGINE_CACHE_ENABLE	0	Add this environment variable if your GPU is **not** T4 or NVIDIA 2080 Ti
KEY_ENV	ASE Encryption key	Add this environment variable if Video_URL is an obfuscated string
IV_ENV	Initialization vector	Add this environment variable if Video_URL is an obfuscated string

IMPORTANT

The `Eula`, `Billing`, and `ApiKey` options must be specified to run the container; otherwise, the container won't start. For more information, see Billing.

Once you update the Deployment manifest for Azure Stack Edge devices, a desktop machine or Azure VM with GPU with your own settings and selection of operations, you can use the below Azure CLI command to deploy the container on the host computer, as an IoT Edge Module.

```
sudo az login
sudo az extension add --name azure-iot
sudo az iot edge set-modules --hub-name "<iothub-name>" --device-id "<device-name>" --content
DeploymentManifest.json --subscription "<name or ID of Azure Subscription>"
```

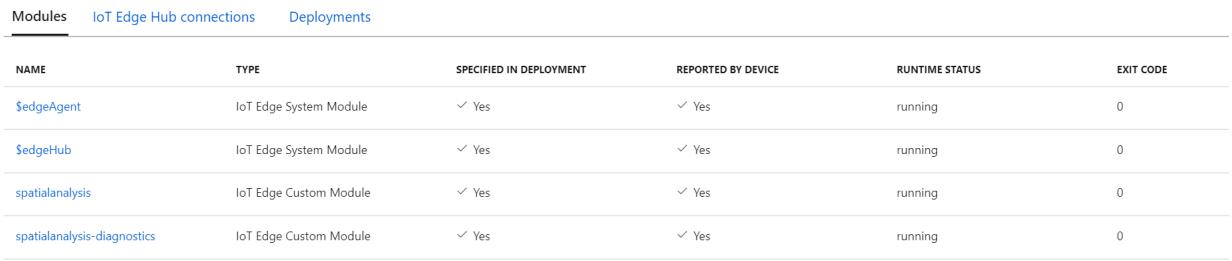
PARAMETER	DESCRIPTION
`--hub-name`	Your Azure IoT Hub name.
`--content`	The name of the deployment file.
`--target-condition`	Your IoT Edge device name for the host computer.
`--subscription`	Subscription ID or name.

This command will start the deployment. Navigate to the page of your Azure IoT Hub instance in the Azure portal to see the deployment status. The status may show as *417 – The device's deployment configuration is not*

set until the device finishes downloading the container images and starts running.

Validate that the deployment is successful

There are several ways to validate that the container is running. Locate the *Runtime Status* in the **IoT Edge Module Settings** for the Spatial Analysis module in your Azure IoT Hub instance on the Azure portal. Validate that the **Desired Value** and **Reported Value** for the *Runtime Status* is *Running*.



Once the deployment is complete and the container is running, the **host computer** will start sending events to the Azure IoT Hub. If you used the `.debug` version of the operations, you'll see a visualizer window for each camera you configured in the deployment manifest. You can now define the lines and zones you want to monitor in the deployment manifest and follow the instructions to deploy again.

Configure the operations performed by Spatial Analysis

You will need to use Spatial Analysis operations to configure the container to use connected cameras, configure the operations, and more. For each camera device you configure, the operations for Spatial Analysis will generate an output stream of JSON messages, sent to your instance of Azure IoT Hub.

Use the output generated by the container

If you want to start consuming the output generated by the container, see the following articles:

- Use the Azure Event Hub SDK for your chosen programming language to connect to the Azure IoT Hub endpoint and receive the events. See Read device-to-cloud messages from the built-in endpoint for more information.
- Set up Message Routing on your Azure IoT Hub to send the events to other endpoints or save the events to Azure Blob Storage, etc. See IoT Hub Message Routing for more information.

Running Spatial Analysis with a recorded video file

You can use Spatial Analysis with both recorded or live video. To use Spatial Analysis for recorded video, try recording a video file and save it as an mp4 file. Create a blob storage account in Azure, or use an existing one. Then update the following blob storage settings in the Azure portal: 1. Change **Secure transfer required** to **Disabled** 2. Change **Allow Blob public access** to **Enabled**

Navigate to the **Container** section, and either create a new container or use an existing one. Then upload the video file to the container. Expand the file settings for the uploaded file, and select **Generate SAS**. Be sure to set the **Expiry Date** long enough to cover the testing period. Set **Allowed Protocols** to *HTTP* (*HTTPS* is not supported).

Click on **Generate SAS Token and URL** and copy the Blob SAS URL. Replace the starting `https` with `http` and test the URL in a browser that supports video playback.

Replace `VIDEO_URL` in the deployment manifest for your Azure Stack Edge device, desktop machine, or Azure VM with GPU with the URL you created, for all of the graphs. Set `VIDEO_IS_LIVE` to `false`, and redeploy the Spatial

Analysis container with the updated manifest. See the example below.

The Spatial Analysis module will start consuming video file and will continuously auto replay as well.

```
"zonecrossing": {
    "operationId" : "cognitiveservices.vision.spatialanalysis-personcrossingpolygon",
    "version": 1,
    "enabled": true,
    "parameters": {
        "VIDEO_URL": "Replace http url here",
        "VIDEO_SOURCE_ID": "personcountgraph",
        "VIDEO_IS_LIVE": false,
      "VIDEO_DECODE_GPU_INDEX": 0,
        "DETECTOR_NODE_CONFIG": "{ \"gpu_index\": 0, \"do_calibration\": true }",
        "SPACEANALYTICS_CONFIG": "{\"zones\":[{\"name\":\"queue\",\"polygon\":[[0.3,0.3],[0.3,0.9],
[0.6,0.9],[0.6,0.3],[0.3,0.3]], \"events\": [{\"type\": \"zonecrossing\", \"config\": {\"threshold\": 16.0,
\"focus\": \"footprint\"}}]}]}"
    }
   },
```

Troubleshooting

If you encounter issues when starting or running the container, see telemetry and troubleshooting for steps for common issues. This article also contains information on generating and collecting logs and collecting system health.

Billing

The Spatial Analysis container sends billing information to Azure, using a Computer Vision resource on your Azure account. The use of Spatial Analysis in public preview is currently free.

Azure Cognitive Services containers aren't licensed to run without being connected to the metering / billing endpoint. You must enable the containers to communicate billing information with the billing endpoint at all times. Cognitive Services containers don't send customer data, such as the video or image that's being analyzed, to Microsoft.

Summary

In this article, you learned concepts and workflow for downloading, installing, and running the Spatial Analysis container. In summary:

- Spatial Analysis is a Linux container for Docker.
- Container images are downloaded from the Microsoft Container Registry.
- Container images run as IoT Modules in Azure IoT Edge.
- How to configure the container and deploy it on a host machine.

Next steps

- Deploy a People Counting web application
- Configure Spatial Analysis operations
- Logging and troubleshooting
- Camera placement guide
- Zone and line placement guide

Spatial Analysis operations

6/28/2021 • 30 minutes to read • Edit Online

Spatial Analysis enables the analysis of real-time streaming video from camera devices. For each camera device you configure, the operations for Spatial Analysis will generate an output stream of JSON messages sent to your instance of Azure IoT Hub.

The Spatial Analysis container implements the following operations:

OPERATION IDENTIFIER	DESCRIPTION
cognitiveservices.vision.spatialanalysis-personcount	Counts people in a designated zone in the camera's field of view. The zone must be fully covered by a single camera in order for PersonCount to record an accurate total. Emits an initial *personCountEvent* event and then *personCountEvent* events when the count changes.
cognitiveservices.vision.spatialanalysis-personcrossingline	Tracks when a person crosses a designated line in the camera's field of view. Emits a *personLineEvent* event when the person crosses the line and provides directional info.
cognitiveservices.vision.spatialanalysis-personcrossingpolygon	Emits a *personZoneEnterExitEvent* event when a person enters or exits the zone and provides directional info with the numbered side of the zone that was crossed. Emits a *personZoneDwellTimeEvent* when the person exits the zone and provides directional info as well as the number of milliseconds the person spent inside the zone.
cognitiveservices.vision.spatialanalysis-persondistance	Tracks when people violate a distance rule. Emits a *personDistanceEvent* periodically with the location of each distance violation.
cognitiveservices.vision.spatialanalysis	Generic operation which can be used to run all scenarios mentioned above. This option is more useful when you want to run multiple scenarios on the same camera or use system resources (e.g. GPU) more efficiently.

All above the operations are also available in the `.debug` version, which have the capability to visualize the video frames as they are being processed. You will need to run `xhost +` on the host computer to enable the visualization of video frames and events.

OPERATION IDENTIFIER	DESCRIPTION
cognitiveservices.vision.spatialanalysis-personcount.debug	Counts people in a designated zone in the camera's field of view. Emits an initial *personCountEvent* event and then *personCountEvent* events when the count changes.
cognitiveservices.vision.spatialanalysis-personcrossingline.debug	Tracks when a person crosses a designated line in the camera's field of view. Emits a *personLineEvent* event when the person crosses the line and provides directional info.

OPERATION IDENTIFIER	DESCRIPTION
cognitiveservices.vision.spatialanalysis-personcrossingpolygon.debug	Emits a *personZoneEnterExitEvent* event when a person enters or exits the zone and provides directional info with the numbered side of the zone that was crossed. Emits a *personZoneDwellTimeEvent* when the person exits the zone and provides directional info as well as the number of milliseconds the person spent inside the zone.
cognitiveservices.vision.spatialanalysis-persondistance.debug	Tracks when people violate a distance rule. Emits a *personDistanceEvent* periodically with the location of each distance violation.
cognitiveservices.vision.spatialanalysis.debug	Generic operation which can be used to run all scenarios mentioned above. This option is more useful when you want to run multiple scenarios on the same camera or use system resources (e.g. GPU) more efficiently.

Spatial Analysis can also be run with Live Video Analytics as their Video AI module.

OPERATION IDENTIFIER	DESCRIPTION
cognitiveservices.vision.spatialanalysis-personcount.livevideoanalytics	Counts people in a designated zone in the camera's field of view. Emits an initial *personCountEvent* event and then *personCountEvent* events when the count changes.
cognitiveservices.vision.spatialanalysis-personcrossingline.livevideoanalytics	Tracks when a person crosses a designated line in the camera's field of view. Emits a *personLineEvent* event when the person crosses the line and provides directional info.
cognitiveservices.vision.spatialanalysis-personcrossingpolygon.livevideoanalytics	Emits a *personZoneEnterExitEvent* event when a person enters or exits the zone and provides directional info with the numbered side of the zone that was crossed. Emits a *personZoneDwellTimeEvent* when the person exits the zone and provides directional info as well as the number of milliseconds the person spent inside the zone.
cognitiveservices.vision.spatialanalysis-persondistance.livevideoanalytics	Tracks when people violate a distance rule. Emits a *personDistanceEvent* periodically with the location of each distance violation.
cognitiveservices.vision.spatialanalysis.livevideoanalytics	Generic operation which can be used to run all scenarios mentioned above. This option is more useful when you want to run multiple scenarios on the same camera or use system resources (e.g. GPU) more efficiently.

Live Video Analytics operations are also available in the `.debug` version (e.g. cognitiveservices.vision.spatialanalysis-personcount.livevideoanalytics.debug) which has the capability to visualize the video frames as being processed. You will need to run `xhost +` on the host computer to enable the visualization of the video frames and events

> **IMPORTANT**
> The computer vision AI models detect and locate human presence in video footage and output by using a bounding box around a human body. The AI models do not attempt to discover the identities or demographics of individuals.

These are the parameters required by each of these Spatial Analysis operations.

OPERATION PARAMETERS	DESCRIPTION
Operation ID	The Operation Identifier from table above.
enabled	Boolean: true or false
VIDEO_URL	The RTSP url for the camera device (Example: `rtsp://username:password@url`). Spatial Analysis supports H.264 encoded stream either through RTSP, http, or mp4. Video_URL can be provided as an obfuscated base64 string value using AES encryption, and if the video url is obfuscated then `KEY_ENV` and `IV_ENV` need to be provided as environment variables. Sample utility to generate keys and encryption can be found here.
VIDEO_SOURCE_ID	A friendly name for the camera device or video stream. This will be returned with the event JSON output.
VIDEO_IS_LIVE	True for camera devices; false for recorded videos.
VIDEO_DECODE_GPU_INDEX	Which GPU to decode the video frame. By default it is 0. Should be the same as the `gpu_index` in other node config like `VICA_NODE_CONFIG`, `DETECTOR_NODE_CONFIG`.
INPUT_VIDEO_WIDTH	Input video/stream's frame width (e.g. 1920). This is an optional field and if provided, the frame will be scaled to this dimension while preserving the aspect ratio.
DETECTOR_NODE_CONFIG	JSON indicating which GPU to run the detector node on. It should be in the following format: `"{ \"gpu_index\": 0 }",`
CAMERA_CONFIG	JSON indicating the calibrated camera parameters for multiple cameras. If the skill you used requires calibration and you already have the camera parameter, you can use this config to provide them directly. Should be in the following format: `"{ \"cameras\": [{\"source_id\": \"endcomputer.0.persondistancegraph.detector+end_computer1\", \"camera_height\": 13.105561256408691, \"camera_focal_length\": 297.60003662109375, \"camera_tiltup_angle\": 0.9738943576812744}] }"`, the `source_id` is used to identify each camera. It can be get from the `source_info` of the event we published. It will only take effect when `do_calibration=false` in `DETECTOR_NODE_CONFIG`.
TRACKER_NODE_CONFIG	JSON indicating whether to compute speed in the tracker node or not. It should be in the following format: `"{ \"enable_speed\": false }",`
SPACEANALYTICS_CONFIG	JSON configuration for zone and line as outlined below.

OPERATION PARAMETERS	DESCRIPTION
ENABLE_FACE_MASK_CLASSIFIER	`True` to enable detecting people wearing face masks in the video stream, `False` to disable it. By default this is disabled. Face mask detection requires input video width parameter to be 1920 `"INPUT_VIDEO_WIDTH": 1920`. The face mask attribute will not be returned if detected people are not facing the camera or are too far from it. Refer to the camera placement guide for more information

Detector Node Parameter Settings

This is an example of the DETECTOR_NODE_CONFIG parameters for all Spatial Analysis operations.

```
{
"gpu_index": 0,
"do_calibration": true,
"enable_recalibration": true,
"calibration_quality_check_frequency_seconds": 86400,
"calibration_quality_check_sample_collect_frequency_seconds": 300,
"calibration_quality_check_one_round_sample_collect_num": 10,
"calibration_quality_check_queue_max_size": 1000,
"calibration_event_frequency_seconds": -1
}
```

NAME	TYPE	DESCRIPTION
`gpu_index`	string	The GPU index on which this operation will run.
`do_calibration`	string	Indicates that calibration is turned on. `do_calibration` must be true for **cognitiveservices.vision.spatialanalysis-persondistance** to function properly. do_calibration is set by default to True.
`enable_recalibration`	bool	Indicates whether automatic recalibration is turned on. Default is `true`.
`calibration_quality_check_frequency_seconds`	int	Minimum number of seconds between each quality check to determine whether or not recalibration is needed. Default is `86400` (24 hours). Only used when `enable_recalibration=True`.
`calibration_quality_check_sample_collect_frequency_seconds`	int	Minimum number of seconds between collecting new data samples for recalibration and quality checking. Default is `300` (5 minutes). Only used when `enable_recalibration=True`.
`calibration_quality_check_one_round_sample_collect_num`	int	Minimum number of new data samples to collect per round of sample collection. Default is `10`. Only used when `enable_recalibration=True`.

NAME	TYPE	DESCRIPTION
`calibration_quality_check_queue_max_size`	int	Maximum number of data samples to store when camera model is calibrated. Default is `1000`. Only used when `enable_recalibration=True`.
`calibration_event_frequency_seconds`	int	Output frequency (seconds) of camera calibration events. A value of `-1` indicates that the camera calibration should not be sent unless the camera calibration info has been changed. Default is `-1`.
`enable_breakpad`	bool	Indicates whether you want to enable breakpad, which is used to generate crash dump for debug use. It is `false` by default. If you set it to `true`, you also need to add `"CapAdd": ["SYS_PTRACE"]` in the `HostConfig` part of container `createOptions`. By default, the crash dump is uploaded to the RealTimePersonTracking AppCenter app, if you want the crash dumps to be uploaded to your own AppCenter app, you can override the environment variable `RTPT_APPCENTER_APP_SECRET` with your app's app secret.
`enable_orientation`	bool	Indicates whether you want to compute the orientation for the detected people or not. `enable_orientation` is set by default to False.

Camera calibration output

This is an example of the output from camera calibration if enabled. Ellipses indicate more of the same type of objects in a list.

```
{
  "type": "cameraCalibrationEvent",
  "sourceInfo": {
    "id": "camera1",
    "timestamp": "2021-04-20T21:15:59.100Z",
    "width": 640,
    "height": 360,
    "frameId": 531,
    "cameraCalibrationInfo": {
      "status": "Calibrated",
      "cameraHeight": 13.294151306152344,
      "focalLength": 372.0000305175781,
      "tiltupAngle": 0.9581864476203918,
      "lastCalibratedTime": "2021-04-20T21:15:59.058"
    }
  },
  "zonePlacementInfo": {
    "optimalZoneRegion": {
      "type": "POLYGON",
       "points": [
        {
```

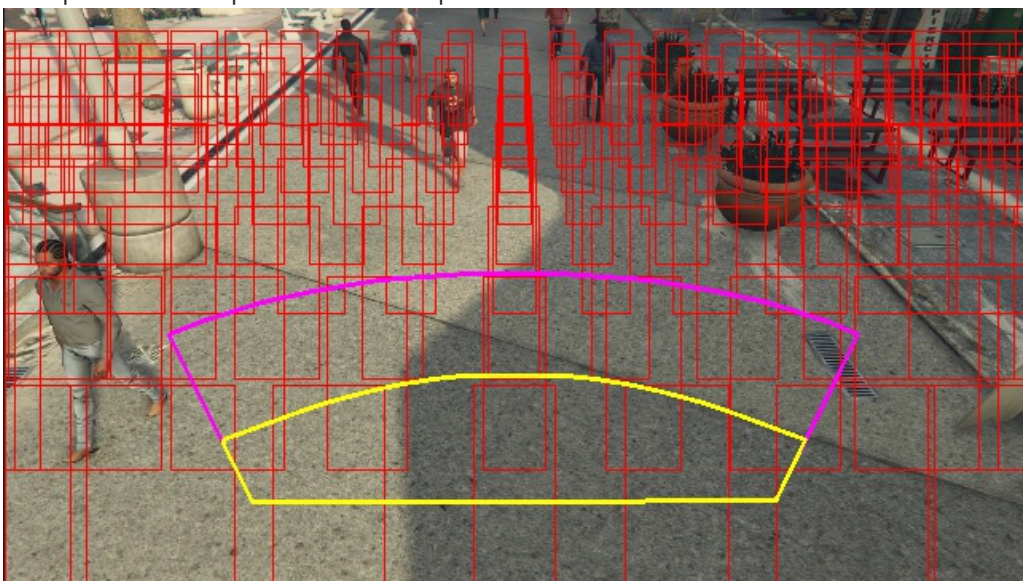
```
            "x": 0.8403755868544601,
            "y": 0.5515320334261838
          },
          {
            "x": 0.15805946791862285,
            "y": 0.5487465181058496
          },
          ...
        ],
        "name": "optimal_zone_region"
      },
      "fairZoneRegion": {
        "type": "POLYGON",
        "points": [
          {
            "x": 0.7871674491392802,
            "y": 0.7437325905292479
          },
          {
            "x": 0.22065727699530516,
            "y": 0.7325905292479109
          },
          ...
        ],
        "name": "fair_zone_region"
      },
      "uniformlySpacedPersonBoundingBoxes": [
        {
          "type": "RECTANGLE",
          "points": [
            {
              "x": 0.0297339593114241,
              "y": 0.0807799442896936
            },
            {
              "x": 0.10015649452269171,
              "y": 0.2757660167130919
            }
          ]
        },
        ...
      ],
      "personBoundingBoxGroundPoints": [
        {
          "x": -22.944068908691406,
          "y": 31.487680435180664
        },
        ...
      ]
    }
  }
```


See Spatial analysis operation output for details on `source_info`.

ZONEPLACEMENTINFO FIELD NAME	TYPE	DESCRIPTION

ZONEPLACEMENTINFO FIELD NAME	TYPE	DESCRIPTION
`optimalZonePolygon`	object	A polygon in the camera image where lines or zones for your operations can be placed for optimal results. Each value pair represents the x,y for vertices of a polygon. The polygon represents the areas in which people are tracked or counted and polygon points are based on normalized coordinates (0-1), where the top left corner is (0.0, 0.0) and the bottom right corner is (1.0, 1.0).
`fairZonePolygon`	object	A polygon in the camera image where lines or zones for your operations can be placed for good, but possibly not optimal, results. See `optimalZonePolygon` above for an in-depth explanation of the contents.
`uniformlySpacedPersonBoundingBoxes`	list	A list of bounding boxes of people within the camera image distributed uniformly in real space. Values are based on normalized coordinates (0-1).
`personBoundingBoxGroundPoints`	list	A list of coordinates on the floor plane relative to the camera. Each coordinate corresponds to the bottom right of the bounding box in `uniformlySpacedPersonBoundingBoxes` with the same index. See the `centerGroundPoint` field under the JSON format for cognitiveservices.vision.spatialanalysis-persondistance AI Insights section for more details on how coordinates on the floor plane are calculated.

Example of the zone placement info output visualized on a video frame:



The zone placement info provides suggestions for your configurations, but the guidelines in Camera configuration must still be followed for best results.

Speed Parameter Settings

You can configure the speed computation through the tracker node parameter settings.

```
{
"enable_speed": true,
}
```

NAME	TYPE	DESCRIPTION
`enable_speed`	bool	Indicates whether you want to compute the speed for the detected people or not. `enable_speed` is set by default to false. It is highly recommended that we enable both speed and orientation to have the best estimated values

Spatial Analysis operations configuration and output

Zone configuration for cognitiveservices.vision.spatialanalysis-personcount

This is an example of a JSON input for the SPACEANALYTICS_CONFIG parameter that configures a zone. You may configure multiple zones for this operation.

```
{
"zones":[{
       "name": "lobbycamera",
       "polygon": [[0.3,0.3], [0.3,0.9], [0.6,0.9], [0.6,0.3], [0.3,0.3]],
       "events":[{
              "type": "count",
              "config":{
                     "trigger": "event",
             "threshold": 16.00,
             "focus": "footprint"
      }
       }]
}
```

NAME	TYPE	DESCRIPTION
`zones`	list	List of zones.
`name`	string	Friendly name for this zone.
`polygon`	list	Each value pair represents the x,y for vertices of a polygon. The polygon represents the areas in which people are tracked or counted and polygon points are based on normalized coordinates (0-1), where the top left corner is (0.0, 0.0) and the bottom right corner is (1.0, 1.0).
`threshold`	float	Events are egressed when the person is greater than this number of pixels inside the zone.

NAME	TYPE	DESCRIPTION
`type`	string	For **cognitiveservices.vision.spatialanalysis-personcount** this should be `count`.
`trigger`	string	The type of trigger for sending an event. Supported values are `event` for sending events when the count changes or `interval` for sending events periodically, irrespective of whether the count has changed or not.
`output_frequency`	int	The rate at which events are egressed. When `output_frequency` = X, every X event is egressed, ex. `output_frequency` = 2 means every other event is output. The `output_frequency` is applicable to both `event` and `interval`.
`focus`	string	The point location within person's bounding box used to calculate events. Focus's value can be `footprint` (the footprint of person), `bottom_center` (the bottom center of person's bounding box), `center` (the center of person's bounding box).

Line configuration for cognitiveservices.vision.spatialanalysis-personcrossingline

This is an example of a JSON input for the SPACEANALYTICS_CONFIG parameter that configures a line. You may configure multiple crossing lines for this operation.

```
{
   "lines": [
       {
           "name": "doorcamera",
           "line": {
               "start": {
                   "x": 0,
                   "y": 0.5
               },
               "end": {
                   "x": 1,
                   "y": 0.5
               }
           },
           "events": [
               {
                   "type": "linecrossing",
                   "config": {
                       "trigger": "event",
                       "threshold": 16.00,
                       "focus": "footprint"
                   }
               }
           ]
       }
   ]
}
```

NAME	TYPE	DESCRIPTION
`lines`	list	List of lines.
`name`	string	Friendly name for this line.
`line`	list	The definition of the line. This is a directional line allowing you to understand "entry" vs. "exit".
`start`	value pair	x, y coordinates for line's starting point. The float values represent the position of the vertex relative to the top,left corner. To calculate the absolute x, y values, you multiply these values with the frame size.
`end`	value pair	x, y coordinates for line's ending point. The float values represent the position of the vertex relative to the top,left corner. To calculate the absolute x, y values, you multiply these values with the frame size.
`threshold`	float	Events are egressed when the person is greater than this number of pixels inside the zone. The default value is 16. This is the recommended value to achieve maximum accuracy.

NAME	TYPE	DESCRIPTION
`type`	string	For **cognitiveservices.vision.spatialanalysis-personcrossingline** this should be `linecrossing`.
`trigger`	string	The type of trigger for sending an event. Supported Values: "event": fire when someone crosses the line.
`focus`	string	The point location within person's bounding box used to calculate events. Focus's value can be `footprint` (the footprint of person), `bottom_center` (the bottom center of person's bounding box), `center` (the center of person's bounding box). The default value is footprint.

Zone configuration for cognitiveservices.vision.spatialanalysis-personcrossingpolygon

This is an example of a JSON input for the SPACEANALYTICS_CONFIG parameter that configures a zone. You may configure multiple zones for this operation.

```
{
"zones":[
  {
      "name": "queuecamera",
      "polygon": [[0.3,0.3], [0.3,0.9], [0.6,0.9], [0.6,0.3], [0.3,0.3]],
      "events":[{
          "type": "zonecrossing",
          "config":{
              "trigger": "event",
              "threshold": 48.00,
              "focus": "footprint"
              }
          }]
  },
  {
      "name": "queuecamera1",
      "polygon": [[0.3,0.3], [0.3,0.9], [0.6,0.9], [0.6,0.3], [0.3,0.3]],
      "events":[{
          "type": "zonedwelltime",
          "config":{
              "trigger": "event",
              "threshold": 16.00,
              "focus": "footprint"
              }
          }]
  }]
}
```

NAME	TYPE	DESCRIPTION
`zones`	list	List of zones.
`name`	string	Friendly name for this zone.

NAME	TYPE	DESCRIPTION
`polygon`	list	Each value pair represents the x,y for vertices of polygon. The polygon represents the areas in which people are tracked or counted. The float values represent the position of the vertex relative to the top,left corner. To calculate the absolute x, y values, you multiply these values with the frame size.
`target_side`	int	Specifies a side of the zone defined by `polygon` to measure how long people face that side while in the zone. 'dwellTimeForTargetSide' will output that estimated time. Each side is a numbered edge between the two vertices of the polygon that represents your zone. For example, the edge between the first two vertices of the polygon represent first side, 'side'=1. The value of `target_side` is between `[0,N-1]` where `N` is the number of sides of the `polygon`. This is an optional field.
`threshold`	float	Events are egressed when the person is greater than this number of pixels inside the zone. The default value is 48 when type is zonecrossing and 16 when time is DwellTime. These are the recommended values to achieve maximum accuracy.
`type`	string	For **cognitiveservices.vision.spatialanalysis-personcrossingpolygon** this should be `zonecrossing` or `zonedwelltime`.
`trigger`	string	The type of trigger for sending an event Supported Values: "event": fire when someone enters or exits the zone.
`focus`	string	The point location within person's bounding box used to calculate events. Focus's value can be `footprint` (the footprint of person), `bottom_center` (the bottom center of person's bounding box), `center` (the center of person's bounding box). The default value is footprint.

Zone configuration for cognitiveservices.vision.spatialanalysis-persondistance

This is an example of a JSON input for the SPACEANALYTICS_CONFIG parameter that configures a zone for **cognitiveservices.vision.spatialanalysis-persondistance**. You may configure multiple zones for this operation.

```
{
"zones":[{
   "name": "lobbycamera",
   "polygon": [[0.3,0.3], [0.3,0.9], [0.6,0.9], [0.6,0.3], [0.3,0.3]],
   "events":[{
       "type": "persondistance",
       "config":{
           "trigger": "event",
           "output_frequency":1,
           "minimum_distance_threshold":6.0,
           "maximum_distance_threshold":35.0,
        "aggregation_method": "average"
           "threshold": 16.00,
           "focus": "footprint"
                   }
         }]
   }]
}
```

NAME	TYPE	DESCRIPTION
`zones`	list	List of zones.
`name`	string	Friendly name for this zone.
`polygon`	list	Each value pair represents the x,y for vertices of polygon. The polygon represents the areas in which people are counted and the distance between people is measured. The float values represent the position of the vertex relative to the top,left corner. To calculate the absolute x, y values, you multiply these values with the frame size.
`threshold`	float	Events are egressed when the person is greater than this number of pixels inside the zone.
`type`	string	For **cognitiveservices.vision.spatialanalysis-persondistance** this should be `people_distance`.
`trigger`	string	The type of trigger for sending an event. Supported values are `event` for sending events when the count changes or `interval` for sending events periodically, irrespective of whether the count has changed or not.
`output_frequency`	int	The rate at which events are egressed. When `output_frequency` = X, every X event is egressed, ex. `output_frequency` = 2 means every other event is output. The `output_frequency` is applicable to both `event` and `interval`.

NAME	TYPE	DESCRIPTION
`minimum_distance_threshold`	float	A distance in feet that will trigger a "TooClose" event when people are less than that distance apart.
`maximum_distance_threshold`	float	A distance in feet that will trigger a "TooFar" event when people are greater than that distance apart.
`aggregation_method`	string	The method for aggregate persondistance result. The aggregation_method is applicable to both `mode` and `average`.
`focus`	string	The point location within person's bounding box used to calculate events. Focus's value can be `footprint` (the footprint of person), `bottom_center` (the bottom center of person's bounding box), `center` (the center of person's bounding box).

Configuration for cognitiveservices.vision.spatialanalysis

This is an example of a JSON input for the SPACEANALYTICS_CONFIG parameter that configures a line and zone for **cognitiveservices.vision.spatialanalysis**. You may configure multiple lines/zones for this operation and each line/zone can have different events.

```
{
 "lines": [
   {
     "name": "doorcamera",
     "line": {
       "start": {
         "x": 0,
         "y": 0.5
       },
       "end": {
         "x": 1,
         "y": 0.5
       }
     },
     "events": [
       {
         "type": "linecrossing",
         "config": {
           "trigger": "event",
           "threshold": 16.00,
           "focus": "footprint"
         }
       }
     ]
   }
 ],
 "zones": [
   {
     "name": "lobbycamera",
     "polygon": [[0.3, 0.3],[0.3, 0.9],[0.6, 0.9],[0.6, 0.3],[0.3, 0.3]],
     "events": [
       {
         "type": "persondistance",
         "config": {
           "trigger": "event"
```

```
            "trigger": "event",
            "output_frequency": 1,
            "minimum_distance_threshold": 6.0,
            "maximum_distance_threshold": 35.0,
            "threshold": 16.00,
            "focus": "footprint"
          }
        },
        {
          "type": "count",
          "config": {
            "trigger": "event",
            "output_frequency": 1,
            "threshold": 16.00,
            "focus": "footprint"
          }
        },
        {
          "type": "zonecrossing",
          "config": {
            "threshold": 48.00,
            "focus": "footprint"
          }
        },
        {
          "type": "zonedwelltime",
          "config": {
            "threshold": 16.00,
            "focus": "footprint"
          }
        }
      ]
    }
  ]
}
```

Camera configuration

See the camera placement guidelines to learn about more about how to configure zones and lines.

Spatial Analysis Operation Output

The events from each operation are egressed to Azure IoT Hub on JSON format.

JSON format for cognitiveservices.vision.spatialanalysis-personcount AI Insights

Sample JSON for an event output by this operation.

```
{
    "events": [
        {
            "id": "b013c2059577418caa826844223bb50b",
            "type": "personCountEvent",
            "detectionIds": [
                "bc796b0fc2534bc59f13138af3dd7027",
                "60add228e5274158897c135905b5a019"
            ],
            "properties": {
                "personCount": 2
            },
            "zone": "lobbycamera",
            "trigger": "event"
        }
    ],
    "sourceInfo": {
        "id": "camera_id",
```

```
        "timestamp": "2020-08-24T06:06:57.224Z",
        "width": 608,
        "height": 342,
        "frameId": "1400",
        "cameraCalibrationInfo": {
            "status": "Calibrated",
            "cameraHeight": 10.306597709655762,
            "focalLength": 385.3199462890625,
            "tiltupAngle": 1.0969393253326416
        },
        "imagePath": ""
    },
    "detections": [
        {
            "type": "person",
            "id": "bc796b0fc2534bc59f13138af3dd7027",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.612683747944079,
                        "y": 0.25340268765276636
                    },
                    {
                        "x": 0.7185954043739721,
                        "y": 0.6425260577285499
                    }
                ]
            },
            "confidence": 0.9559211134910583,
            "centerGroundPoint": {
                "x": 0.0,
                "y": 0.0
            },
            "metadata": {
            "attributes": {
                "face_mask": 0.99
            }
        }
        },
        {
            "type": "person",
            "id": "60add228e5274158897c135905b5a019",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.22326200886776573,
                        "y": 0.17830915618361087
                    },
                    {
                        "x": 0.34922296122500773,
                        "y": 0.6297955429344847
                    }
                ]
            },
            "confidence": 0.9389744400978088,
            "centerGroundPoint": {
                "x": 0.0,
                "y": 0.0
            },
            "metadata":{
            "attributes": {
            "face_nomask": 0.99
            }
            }
       }
    ],
    "schemaVersion": "1.0"
```

```
}
```

EVENT FIELD NAME	TYPE	DESCRIPTION
`id`	string	Event ID
`type`	string	Event type
`detectionsId`	array	Array of size 1 of unique identifier of the person detection that triggered this event
`properties`	collection	Collection of values
`trackinId`	string	Unique identifier of the person detected
`zone`	string	The "name" field of the polygon that represents the zone that was crossed
`trigger`	string	The trigger type is 'event' or 'interval' depending on the value of `trigger` in SPACEANALYTICS_CONFIG

DETECTIONS FIELD NAME	TYPE	DESCRIPTION
`id`	string	Detection ID
`type`	string	Detection type
`region`	collection	Collection of values
`type`	string	Type of region
`points`	collection	Top left and bottom right points when the region type is RECTANGLE
`confidence`	float	Algorithm confidence
`face_mask`	float	The attribute confidence value with range (0-1) indicates the detected person is wearing a face mask
`face_nomask`	float	The attribute confidence value with range (0-1) indicates the detected person is **not** wearing a face mask

SOURCEINFO FIELD NAME	TYPE	DESCRIPTION
`id`	string	Camera ID
`timestamp`	date	UTC date when the JSON payload was emitted

SOURCEINFO FIELD NAME	TYPE	DESCRIPTION
`width`	int	Video frame width
`height`	int	Video frame height
`frameId`	int	Frame identifier
`cameraCallibrationInfo`	collection	Collection of values
`status`	string	The status of the calibration in the format of `state[;progress description]`. The state can be `Calibrating`, `Recalibrating` (if recalibration is enabled), or `Calibrated`. The progress description part is only valid when it is in `Calibrating` and `Recalibrating` state, which is used to show the progress of current calibration process.
`cameraHeight`	float	The height of the camera above the ground in feet. This is inferred from auto-calibration.
`focalLength`	float	The focal length of the camera in pixels. This is inferred from auto-calibration.
`tiltUpAngle`	float	The camera tilt angle from vertical. This is inferred from auto-calibration.

JSON format for cognitiveservices.vision.spatialanalysis-personcrossingline AI Insights

Sample JSON for detections output by this operation.

```
{
    "events": [
        {
            "id": "3733eb36935e4d73800a9cf36185d5a2",
            "type": "personLineEvent",
            "detectionIds": [
                "90d55bfc64c54bfd98226697ad8445ca"
            ],
            "properties": {
                "trackingId": "90d55bfc64c54bfd98226697ad8445ca",
                "status": "CrossLeft"
            },
            "zone": "doorcamera"
        }
    ],
    "sourceInfo": {
        "id": "camera_id",
        "timestamp": "2020-08-24T06:06:53.261Z",
        "width": 608,
        "height": 342,
        "frameId": "1340",
        "imagePath": ""
    },
    "detections": [
        {
            "type": "person",
            "id": "90d55bfc64c54bfd98226697ad8445ca",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.491627341822574,
                        "y": 0.2385801348769874
                    },
                    {
                        "x": 0.588894994635331,
                        "y": 0.6395559924387793
                    }
                ]
            },
            "confidence": 0.9005028605461121,
            "metadata": {
            "attributes": {
                "face_mask": 0.99
            }
        }
        }
    ],
    "schemaVersion": "1.0"
}
```


EVENT FIELD NAME	TYPE	DESCRIPTION
`id`	string	Event ID
`type`	string	Event type
`detectionsId`	array	Array of size 1 of unique identifier of the person detection that triggered this event
`properties`	collection	Collection of values

EVENT FIELD NAME	TYPE	DESCRIPTION
`trackinId`	string	Unique identifier of the person detected
`status`	string	Direction of line crossings, either 'CrossLeft' or 'CrossRight'. Direction is based on imagining standing at the "start" facing the "end" of the line. CrossRight is crossing from left to right. CrossLeft is crossing from right to left.
`orientationDirection`	string	The orientation direction of the detected person after crossing the line. The value can be 'Left', 'Right, or 'Straight'. This value is output if `enable_orientation` is set to `True` in `DETECTOR_NODE_CONFIG`
`zone`	string	The "name" field of the line that was crossed

DETECTIONS FIELD NAME	TYPE	DESCRIPTION
`id`	string	Detection ID
`type`	string	Detection type
`region`	collection	Collection of values
`type`	string	Type of region
`points`	collection	Top left and bottom right points when the region type is RECTANGLE
`groundOrientationAngle`	float	The clockwise radian angle of the person's orientation on the inferred ground plane
`mappedImageOrientation`	float	The projected clockwise radian angle of the person's orientation on the 2D image space
`speed`	float	The estimated speed of the detected person. The unit is `foot per second (ft/s)`
`confidence`	float	Algorithm confidence
`face_mask`	float	The attribute confidence value with range (0-1) indicates the detected person is wearing a face mask

DETECTIONS FIELD NAME	TYPE	DESCRIPTION
`face_nomask`	float	The attribute confidence value with range (0-1) indicates the detected person is **not** wearing a face mask

SOURCEINFO FIELD NAME	TYPE	DESCRIPTION
`id`	string	Camera ID
`timestamp`	date	UTC date when the JSON payload was emitted
`width`	int	Video frame width
`height`	int	Video frame height
`frameId`	int	Frame identifier

> **IMPORTANT**
> The AI model detects a person irrespective of whether the person is facing towards or away from the camera. The AI model doesn't run face recognition and doesn't emit any biometric information.

JSON format for cognitiveservices.vision.spatialanalysis-personcrossingpolygon AI Insights

Sample JSON for detections output by this operation with `zonecrossing` type SPACEANALYTICS_CONFIG.

```
{
    "events": [
        {
            "id": "f095d6fe8cfb4ffaa8c934882fb257a5",
            "type": "personZoneEnterExitEvent",
            "detectionIds": [
                "afcc2e2a32a6480288e24381f9c5d00e"
            ],
            "properties": {
                "trackingId": "afcc2e2a32a6480288e24381f9c5d00e",
                "status": "Enter",
                "side": "1"
            },
            "zone": "queuecamera"
        }
    ],
    "sourceInfo": {
        "id": "camera_id",
        "timestamp": "2020-08-24T06:15:09.680Z",
        "width": 608,
        "height": 342,
        "frameId": "428",
        "imagePath": ""
    },
    "detections": [
        {
            "type": "person",
            "id": "afcc2e2a32a6480288e24381f9c5d00e",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.8135572734631991,
                        "y": 0.6653949670624315
                    },
                    {
                        "x": 0.9937645761590255,
                        "y": 0.9925406829655519
                    }
                ]
            },
            "confidence": 0.6267998814582825,
        "metadata": {
        "attributes": {
        "face_mask": 0.99
        }
        }

        }
    ],
    "schemaVersion": "1.0"
}
```


Sample JSON for detections output by this operation with `zonedwelltime` type SPACEANALYTICS_CONFIG.

```
{
    "events": [
        {
            "id": "f095d6fe8cfb4ffaa8c934882fb257a5",
            "type": "personZoneDwellTimeEvent",
            "detectionIds": [
                "afcc2e2a32a6480288e24381f9c5d00e"
            ],
            "properties": {
                "trackingId": "afcc2e2a32a6480288e24381f9c5d00e",
                "status": "Exit",
                "side": "1",
                "dwellTime": 7132.0,
                "dwellFrames": 20
            },
            "zone": "queuecamera"
        }
    ],
    "sourceInfo": {
        "id": "camera_id",
        "timestamp": "2020-08-24T06:15:09.680Z",
        "width": 608,
        "height": 342,
        "frameId": "428",
        "imagePath": ""
    },
    "detections": [
        {
            "type": "person",
            "id": "afcc2e2a32a6480288e24381f9c5d00e",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.8135572734631991,
                        "y": 0.6653949670624315
                    },
                    {
                        "x": 0.9937645761590255,
                        "y": 0.9925406829655519
                    }
                ]
            },
            "confidence": 0.6267998814582825,
            "metadataType": "",
          "metadata": {
               "groundOrientationAngle": 1.2,
               "mappedImageOrientation": 0.3,
               "speed": 1.2
            },
        }
    ],
    "schemaVersion": "1.0"
}
```


EVENT FIELD NAME	TYPE	DESCRIPTION
`id`	string	Event ID
`type`	string	Event type. The value can be either *personZoneDwellTimeEvent* or *personZoneEnterExitEvent*

EVENT FIELD NAME	TYPE	DESCRIPTION
`detectionsId`	array	Array of size 1 of unique identifier of the person detection that triggered this event
`properties`	collection	Collection of values
`trackinId`	string	Unique identifier of the person detected
`status`	string	Direction of polygon crossings, either 'Enter' or 'Exit'
`side`	int	The number of the side of the polygon that the person crossed. Each side is a numbered edge between the two vertices of the polygon that represents your zone. The edge between the first two vertices of the polygon represent first side. 'Side' is empty when the event isn't associated with a specific side due to occlusion. For example, an exit occurred when a person disappears but wasn't seen crossing a side of the zone or an enter occurred when a person appeared in the zone but wasn't seen crossing a side.
`dwellTime`	float	The number of milliseconds that represent the time the person spent in the zone. This field is provided when the event type is personZoneDwellTimeEvent
`dwellFrames`	int	The number of frames that the person spent in the zone. This field is provided when the event type is personZoneDwellTimeEvent
`dwellTimeForTargetSide`	float	The number of milliseconds that represent the time the person spent in the zone and were facing to the `target_side`. This field is provided when `enable_orientation` is `True` in `DETECTOR_NODE_CONFIG` and the value of `target_side` is set in `SPACEANALYTICS_CONFIG`
`avgSpeed`	float	The average speed of the person in the zone. The unit is `foot per second (ft/s)`
`minSpeed`	float	The minimum speed of the person in the zone. The unit is `foot per second (ft/s)`

EVENT FIELD NAME	TYPE	DESCRIPTION
`zone`	string	The "name" field of the polygon that represents the zone that was crossed

DETECTIONS FIELD NAME	TYPE	DESCRIPTION
`id`	string	Detection ID
`type`	string	Detection type
`region`	collection	Collection of values
`type`	string	Type of region
`points`	collection	Top left and bottom right points when the region type is RECTANGLE
`groundOrientationAngle`	float	The clockwise radian angle of the person's orientation on the inferred ground plane
`mappedImageOrientation`	float	The projected clockwise radian angle of the person's orientation on the 2D image space
`speed`	float	The estimated speed of the detected person. The unit is `foot per second (ft/s)`
`confidence`	float	Algorithm confidence
`face_mask`	float	The attribute confidence value with range (0-1) indicates the detected person is wearing a face mask
`face_nomask`	float	The attribute confidence value with range (0-1) indicates the detected person is **not** wearing a face mask

JSON format for cognitiveservices.vision.spatialanalysis-persondistance AI Insights

Sample JSON for detections output by this operation.


```
{
    "events": [
        {
            "id": "9c15619926ef417aa93c1faf00717d36",
            "type": "personDistanceEvent",
            "detectionIds": [
                "9037c65fa3b74070869ee5110fcd23ca",
                "7ad7f43fd1a64971ae1a30dbeeffc38a"
            ],
            "properties": {
                "personCount": 5,
                "averageDistance": 20.807043981552123,
                "minimumDistanceThreshold": 6.0,
                "maximumDistanceThreshold": "Infinity",
```

```
                "eventName": "TooClose",
                "distanceViolationPersonCount": 2
            },
            "zone": "lobbycamera",
            "trigger": "event"
        }
    ],
    "sourceInfo": {
        "id": "camera_id",
        "timestamp": "2020-08-24T06:17:25.309Z",
        "width": 608,
        "height": 342,
        "frameId": "1199",
        "cameraCalibrationInfo": {
            "status": "Calibrated",
            "cameraHeight": 12.9940824508667,
            "focalLength": 401.2800598144531,
            "tiltupAngle": 1.057669997215271
        },
        "imagePath": ""
    },
    "detections": [
        {
            "type": "person",
            "id": "9037c65fa3b74070869ee5110fcd23ca",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.39988183975219727,
                        "y": 0.2719132942065858
                    },
                    {
                        "x": 0.5051516984638414,
                        "y": 0.6488402517218339
                    }
                ]
            },
            "confidence": 0.948630690574646,
            "centerGroundPoint": {
                "x": -1.4638760089874268,
                "y": 18.29732322692871
            },
            "metadataType": ""
        },
        {
            "type": "person",
            "id": "7ad7f43fd1a64971ae1a30dbeeffc38a",
            "region": {
                "type": "RECTANGLE",
                "points": [
                    {
                        "x": 0.5200299714740954,
                        "y": 0.2875368218672903
                    },
                    {
                        "x": 0.6457497446160567,
                        "y": 0.6183311060855263
                    }
                ]
            },
            "confidence": 0.8235412240028381,
            "centerGroundPoint": {
                "x": 2.6310102939605713,
                "y": 18.635927200317383
            },
            "metadataType": ""
        }
    ],
    "schemaVersion": "1.0"
```

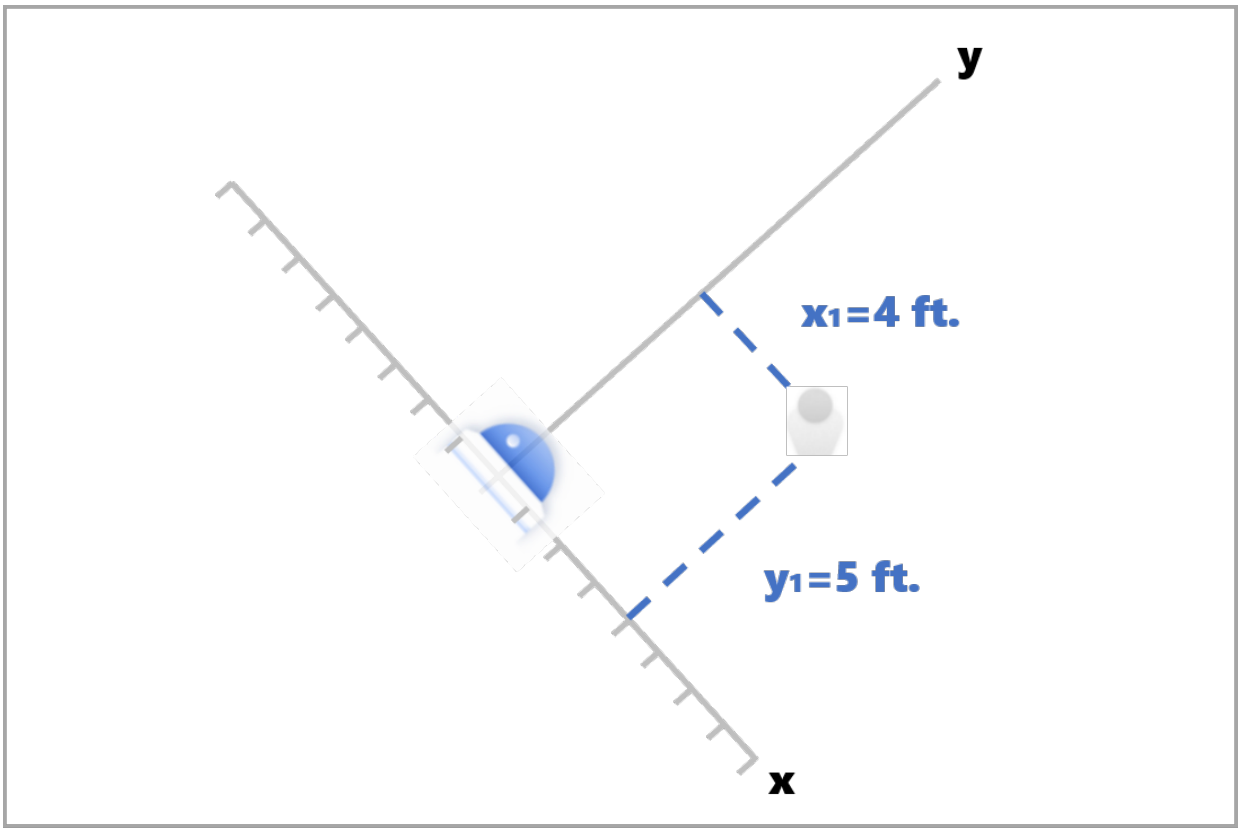
```
    schemaVersion : "1.0"
}
```

EVENT FIELD NAME	TYPE	DESCRIPTION
`id`	string	Event ID
`type`	string	Event type
`detectionsId`	array	Array of size 1 of unique identifier of the person detection that triggered this event
`properties`	collection	Collection of values
`personCount`	int	Number of people detected when the event was emitted
`averageDistance`	float	The average distance between all detected people in feet
`minimumDistanceThreshold`	float	The distance in feet that will trigger a "TooClose" event when people are less than that distance apart.
`maximumDistanceThreshold`	float	The distance in feet that will trigger a "TooFar" event when people are greater than distance apart.
`eventName`	string	Event name is `TooClose` with the `minimumDistanceThreshold` is violated, `TooFar` when `maximumDistanceThreshold` is violated, or `unknown` when auto-calibration hasn't completed
`distanceViolationPersonCount`	int	Number of people detected in violation of `minimumDistanceThreshold` or `maximumDistanceThreshold`
`zone`	string	The "name" field of the polygon that represents the zone that was monitored for distancing between people
`trigger`	string	The trigger type is 'event' or 'interval' depending on the value of `trigger` in SPACEANALYTICS_CONFIG

DETECTIONS FIELD NAME	TYPE	DESCRIPTION
`id`	string	Detection ID
`type`	string	Detection type

DETECTIONS FIELD NAME	TYPE	DESCRIPTION
`region`	collection	Collection of values
`type`	string	Type of region
`points`	collection	Top left and bottom right points when the region type is RECTANGLE
`confidence`	float	Algorithm confidence
`centerGroundPoint`	2 float values	`x`, `y` values with the coordinates of the person's inferred location on the ground in feet. `x` and `y` are coordinates on the floor plane, assuming the floor is level. The camera's location is the origin.

When calculating `centerGroundPoint`, `x` is the distance from the camera to the person along a line perpendicular to the camera image plane. `y` is the distance from the camera to the person along a line parallel to the camera image plane.



In this example, `centerGroundPoint` is `{x: 4, y: 5}`. This means there's a person 4 feet away from the camera and 5 feet to the right, looking at the room top-down.

SOURCEINFO FIELD NAME	TYPE	DESCRIPTION
`id`	string	Camera ID
`timestamp`	date	UTC date when the JSON payload was emitted

SOURCEINFO FIELD NAME	TYPE	DESCRIPTION
`width`	int	Video frame width
`height`	int	Video frame height
`frameId`	int	Frame identifier
`cameraCallibrationInfo`	collection	Collection of values
`status`	string	The status of the calibration in the format of `state[;progress description]`. The state can be `Calibrating`, `Recalibrating` (if recalibration is enabled), or `Calibrated`. The progress description part is only valid when it is in `Calibrating` and `Recalibrating` state, which is used to show the progress of current calibration process.
`cameraHeight`	float	The height of the camera above the ground in feet. This is inferred from auto-calibration.
`focalLength`	float	The focal length of the camera in pixels. This is inferred from auto-calibration.
`tiltUpAngle`	float	The camera tilt angle from vertical. This is inferred from auto-calibration.

JSON format for cognitiveservices.vision.spatialanalysis AI Insights

Output of this operation depends on configured `events`, for example if the there is a `zonecrossing` event configured for this operation then output will be same as `cognitiveservices.vision.spatialanalysis-personcrossingpolygon`.

Use the output generated by the container

You may want to integrate Spatial Analysis detection or events into your application. Here are a few approaches to consider:

- Use the Azure Event Hub SDK for your chosen programming language to connect to the Azure IoT Hub endpoint and receive the events. See Read device-to-cloud messages from the built-in endpoint for more information.
- Set up **Message Routing** on your Azure IoT Hub to send the events to other endpoints or save the events to your data storage. See IoT Hub Message Routing for more information.
- Setup an Azure Stream Analytics job to process the events in real-time as they arrive and create visualizations.

Deploying Spatial Analysis operations at scale (multiple cameras)

In order to get the best performance and utilization of the GPUs, you can deploy any Spatial Analysis operations on multiple cameras using graph instances. Below is a sample for running the

`cognitiveservices.vision.spatialanalysis-personcrossingline` operation on fifteen cameras.

```
  "properties.desired": {
      "globalSettings": {
          "PlatformTelemetryEnabled": false,
          "CustomerTelemetryEnabled": true
      },
      "graphs": {
        "personzonelinecrossing": {
        "operationId": "cognitiveservices.vision.spatialanalysis-personcrossingline",
        "version": 1,
        "enabled": true,
        "sharedNodes": {
            "shared_detector0": {
                "node": "PersonCrossingLineGraph.detector",
                "parameters": {
                    "DETECTOR_NODE_CONFIG": "{ \"gpu_index\": 0, \"batch_size\": 7, \"do_calibration\":
true}",
                }
            },
            "shared_detector1": {
                "node": "PersonCrossingLineGraph.detector",
                "parameters": {
                    "DETECTOR_NODE_CONFIG": "{ \"gpu_index\": 0, \"batch_size\": 8, \"do_calibration\":
true}",
                }
            }
        },
        "parameters": {
            "VIDEO_DECODE_GPU_INDEX": 0,
            "VIDEO_IS_LIVE": true
        },
        "instances": {
            "1": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector0",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 1>",
                    "VIDEO_SOURCE_ID": "camera 1",
                    "SPACEANALYTICS_CONFIG": "{\"zones\":[{\"name\":\"queue\",\"polygon\":[[0,0],[1,0],
[0,1],[1,1],[0,0]]}]}"
                }
            },
            "2": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector0",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 2>",
                    "VIDEO_SOURCE_ID": "camera 2",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            },
            "3": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector0",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 3>",
                    "VIDEO_SOURCE_ID": "camera 3",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            },
            "4": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector0",
                },
```

```
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 4>",
                "VIDEO_SOURCE_ID": "camera 4",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "5": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector0",
            },
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 5>",
                "VIDEO_SOURCE_ID": "camera 5",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "6": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector0",
            },
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 6>",
                "VIDEO_SOURCE_ID": "camera 6",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "7": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector0",
            },
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 7>",
                "VIDEO_SOURCE_ID": "camera 7",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "8": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector1",
            },
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 8>",
                "VIDEO_SOURCE_ID": "camera 8",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "9": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector1",
            },
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 9>",
                "VIDEO_SOURCE_ID": "camera 9",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "10": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector1",
            },
            "parameters": {
                "VIDEO_URL": "<Replace RTSP URL for camera 10>",
                "VIDEO_SOURCE_ID": "camera 10",
                "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
            }
        },
        "11": {
            "sharedNodeMap": {
                "PersonCrossingLineGraph/detector": "shared_detector1",
```

```
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 11>",
                    "VIDEO_SOURCE_ID": "camera 11",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            },
            "12": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector1",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 12>",
                    "VIDEO_SOURCE_ID": "camera 12",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            },
            "13": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector1",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 13>",
                    "VIDEO_SOURCE_ID": "camera 13",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            },
            "14": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector1",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 14>",
                    "VIDEO_SOURCE_ID": "camera 14",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            },
            "15": {
                "sharedNodeMap": {
                    "PersonCrossingLineGraph/detector": "shared_detector1",
                },
                "parameters": {
                    "VIDEO_URL": "<Replace RTSP URL for camera 15>",
                    "VIDEO_SOURCE_ID": "camera 15",
                    "SPACEANALYTICS_CONFIG": "<Replace the zone config value, same format as above>"
                }
            }
          }
        },
      }
  }
```

NAME	TYPE	DESCRIPTION
`batch_size`	int	If all of the cameras have the same resolution, set `batch_size` to the number of cameras that will be used in that operation, otherwise, set `batch_size` to 1 or leave it as default (1), which indicates no batch is supported.

Next steps

- Deploy a People Counting web application

- Logging and troubleshooting
- Camera placement guide
- Zone and line placement guide

How to: Deploy a Spatial Analysis web application

6/8/2021 • 6 minutes to read • Edit Online

Use this article to learn how to deploy a web app which will collect spatial analysis data(insights) from IotHub and visualize it. This can have useful applications across a wide range of scenarios and industries. For example, if a company wants to optimize the use of its real estate space, they are able to quickly create a solution with different scenarios.

In this tutorial you will learn how to:

- Deploy the Spatial Analysis container
- Configure the operation and camera
- Configure the IoT Hub connection in the Web Application
- Deploy and test the Web Application

This app will showcase below scenarios:

- Count of people entering and exiting a space/store
- Count of people entering and exiting a checkout area/zone and the time spent in the checkout line (dwell time)
- Count of people wearing a face mask
- Count of people violating social distancing guidelines

Prerequisites

- Azure subscription - create one for free
- Basic understanding of Azure IoT Edge deployment configurations, and an Azure IoT Hub
- A configured host computer.

Deploy the Spatial Analysis container

Fill out the request application to get access to run the container.

Follow the Host Computer Setup to configure the host computer and connect an IoT Edge device to Azure IoT Hub.

Deploy an Azure IoT Hub service in your subscription

First, create an instance of an Azure IoT Hub service with either the Standard Pricing Tier (S1) or Free Tier (S0). Follow these instructions to create this instance using the Azure CLI.

Fill in the required parameters:

- Subscription: The name or ID of your Azure Subscription
- Resource group: Create a name for your resource group
- Iot Hub Name: Create a name for your IoT Hub
- IoTHub Name: The name of the IoT Hub you created
- Edge Device Name: Create a name for your Edge Device

```
az login
az account set --subscription <name or ID of Azure Subscription>
az group create --name "<Resource Group Name>" --location "WestUS"

az iot hub create --name "<IoT Hub Name>" --sku S1 --resource-group "test-resource-group"

az iot hub device-identity create --hub-name "<IoT Hub Name>" --device-id "<Edge Device Name>" --edge-
enabled
```

Deploy the container on Azure IoT Edge on the host computer

The next step is to deploy the **spatial analysis** container as an IoT Module on the host computer using the Azure CLI. The deployment process requires a Deployment Manifest file which outlines the required containers, variables, and configurations for your deployment. A sample Deployment Manifest can be found at DeploymentManifest.json which includes pre-built configurations for all scenarios.

Set environment variables

Most of the **Environment Variables** for the IoT Edge Module are already set in the sample *DeploymentManifest.json* files linked above. In the file, search for the `ENDPOINT` and `APIKEY` environment variables, shown below. Replace the values with the Endpoint URI and the API Key that you created earlier. Ensure that the EULA value is set to "accept".

```
"EULA": {
    "value": "accept"
},
"ENDPOINT":{
    "value": "<Use a key from your Computer Vision resource>"
},
"APIKEY":{
    "value": "<Use the endpoint from your Computer Vision resource>"
}
```

Configure the operation parameters

If you are using the sample DeploymentManifest.json which already has all of the required configurations (operations, recorded video file urls and zones etc.), then you can skip to the **Execute the deployment** section.

Now that the initial configuration of the spatial analysis container is complete, the next step is to configure the operations parameters and add them to the deployment.

The first step is to update the sample DeploymentManifest.json and configure the desired operation. For example, configuration for cognitiveservices.vision.spatialanalysis-personcount is shown below:

```
"personcount": {
    "operationId": "cognitiveservices.vision.spatialanalysis-personcount",
    "version": 1,
    "enabled": true,
    "parameters": {
        "VIDEO_URL": "<Replace RTSP URL here>",
        "VIDEO_SOURCE_ID": "<Replace with friendly name>",
        "VIDEO_IS_LIVE":true,
        "DETECTOR_NODE_CONFIG": "{ \"gpu_index\": 0 }",
        "SPACEANALYTICS_CONFIG": "{\"zones\":[{\"name\":\"queue\",\"polygon\":[<Replace with your values>],
\"events\": [{\"type\":\"count\"}], \"threshold\":<use 0 for no threshold.}]}"
    }
},
```

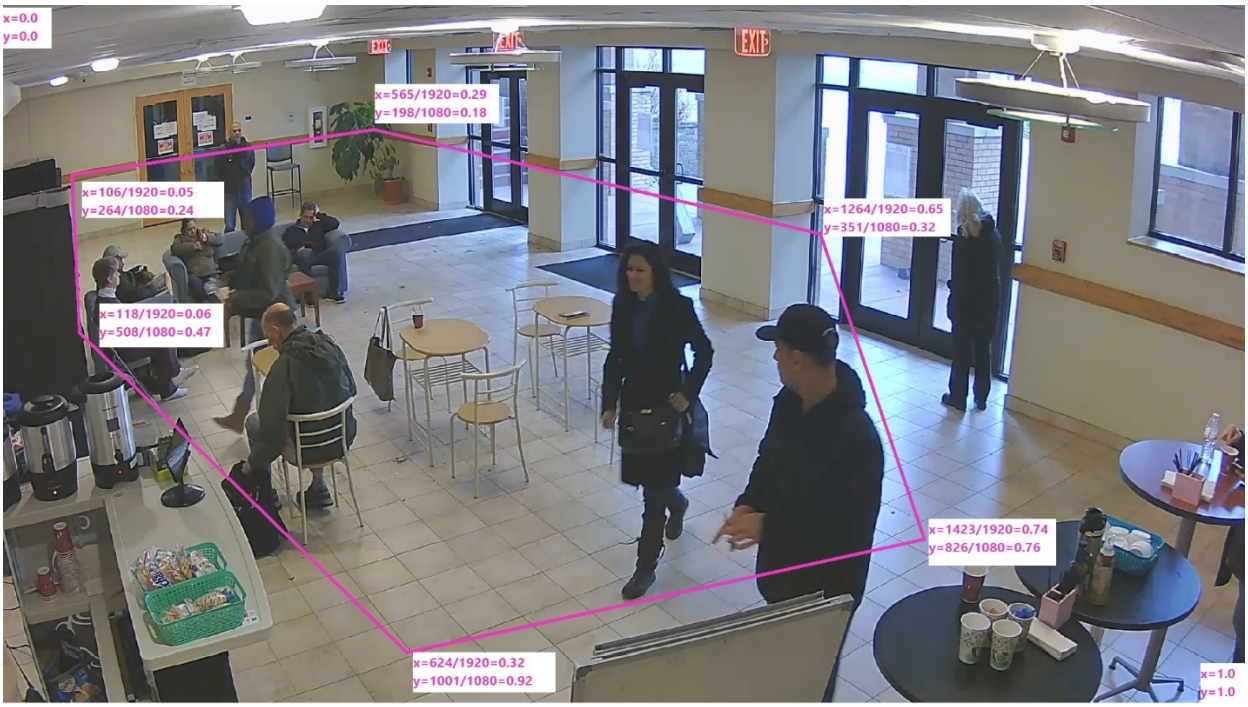
After the deployment manifest is updated, follow the camera manufacturer's instructions to install the camera, configure the camera url, and configure the user name and password.

Next, set `VIDEO_URL` to the RTSP url of the camera, and the credentials for connecting to the camera.

If the edge device has more than one GPU, select the GPU on which to run this operation. Make sure you load balance the operations where there are no more than 8 operations running on a single GPU at a time.

Next, configure the zone in which you want to count people. To configure the zone polygon, first follow the manufacturer's instructions to retrieve a frame from the camera. To determine each vertex of the polygon, select a point on the frame, take the x,y pixel coordinates of the point relative to the left, top corner of the frame, and divide by the corresponding frame dimensions. Set the results as x,y coordinates of the vertex. You can set the zone polygon configuration in the `SPACEANALYTICS_CONFIG` field.

This is a sample video frame that shows how the vertex coordinates are being calculated for a frame of size 1920/1080.



You can also select a confidence threshold for when detected people are counted and events are generated. Set the threshold to 0 if you'd like all events to be output.

Execute the deployment

Now that the deployment manifest is complete, use this command in the Azure CLI to deploy the container on the host computer as an IoT Edge Module.

```
az login
az extension add --name azure-iot
az iot edge set-modules --hub-name "<IoT Hub name>" --device-id "<IoT Edge device name>" --content
DeploymentManifest.json --subscription "<subscriptionId>"
```

Fill in the required parameters:

- IoT Hub Name: Your Azure IoT Hub name
- DeploymentManifest.json: The name of your deployment file
- IoT Edge device name: The IoT Edge device name of your host computer
- Subscription: Your subscription ID or name

This command will begin the deployment, and you can view the deployment status in your Azure IoT Hub instance in the Azure portal. The status may show as *417 – The device's deployment configuration is not set* until the device finishes downloading the container images and starts running.

Validate that the deployment was successful

Locate the *Runtime Status* in the IoT Edge Module Settings for the spatial-analysis module in your IoT Hub instance on the Azure portal. The **Desired Value** and **Reported Value** for the *Runtime Status* should say `Running`. See below for what this will look like on the Azure portal.

Modules | IoT Edge Hub connections | Deployments

NAME	TYPE	SPECIFIED IN DEPLOYMENT	REPORTED BY DEVICE	RUNTIME STATUS	EXIT CODE
$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes	running	0
$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes	running	0
spatialanalysis	IoT Edge Custom Module	✓ Yes	✓ Yes	running	0
spatialanalysis-diagnostics	IoT Edge Custom Module	✓ Yes	✓ Yes	running	0

At this point, the spatial analysis container is running the operation. It emits AI insights for the operations and routes these insights as telemetry to your Azure IoT Hub instance. To configure additional cameras, you can update the deployment manifest file and execute the deployment again.

Spatial Analysis Web Application

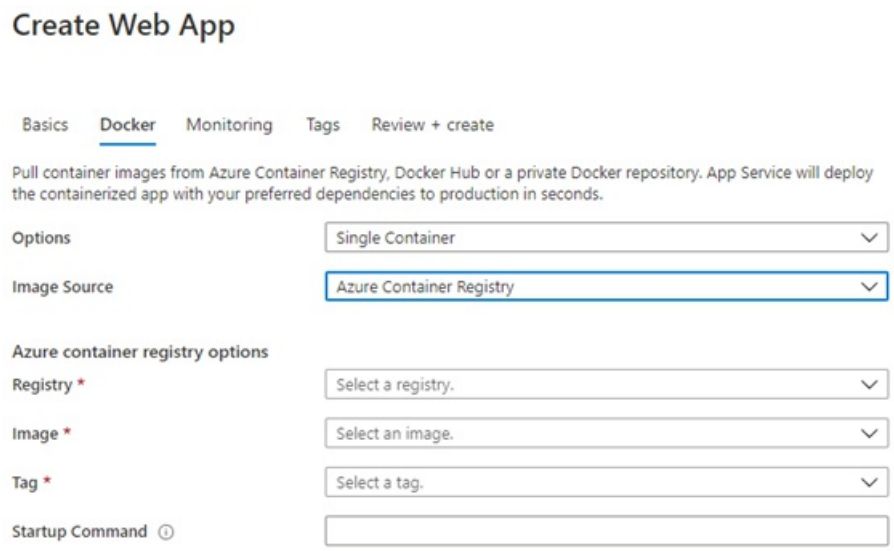
The Spatial Analysis Web Application enables developers to quickly configure a sample web app, host it in their Azure environment, and use the app to validate E2E events.

Build Docker Image

Follow the guide to build and push the image to an Azure Container Registry in your subscription.

Setup Steps

To install the container, create a new Azure App Service and fill in the required parameters. Then go to the **Docker** Tab and select **Single Container**, then **Azure Container Registry**. Use your instance of Azure Container Registry where you pushed the image above.



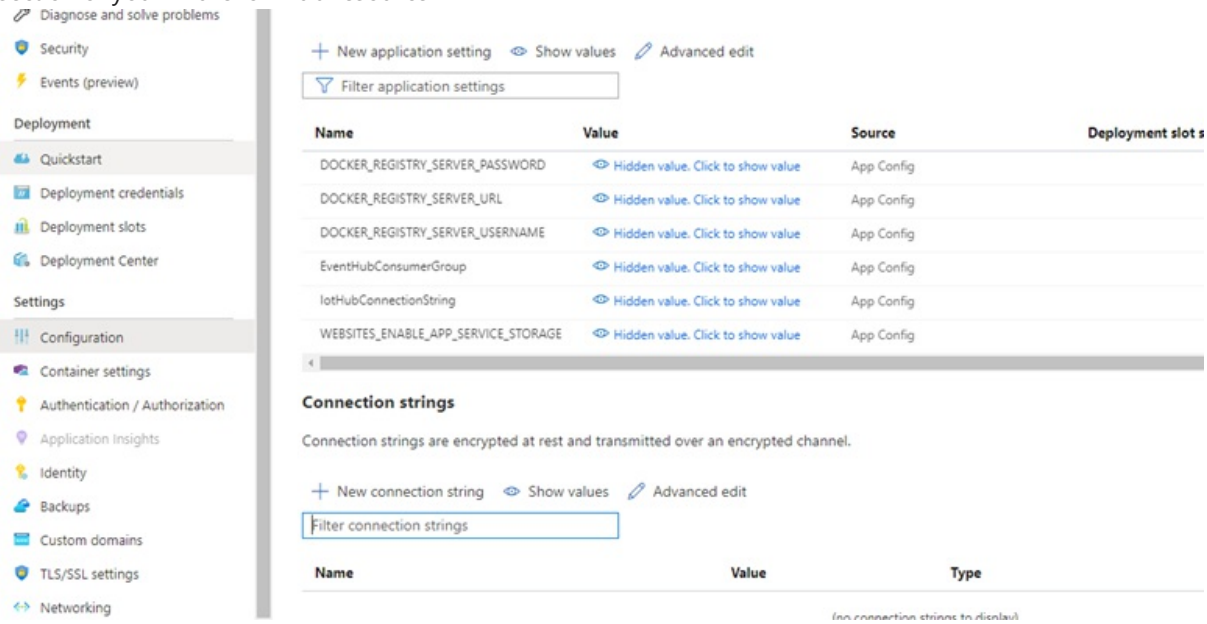
After entering the above parameters, click on **Review+Create** and create the app.

Configure the app

Wait for setup to complete, and navigate to your resource in the Azure portal. Go to the **configuration** section

and add the following two **application settings**.

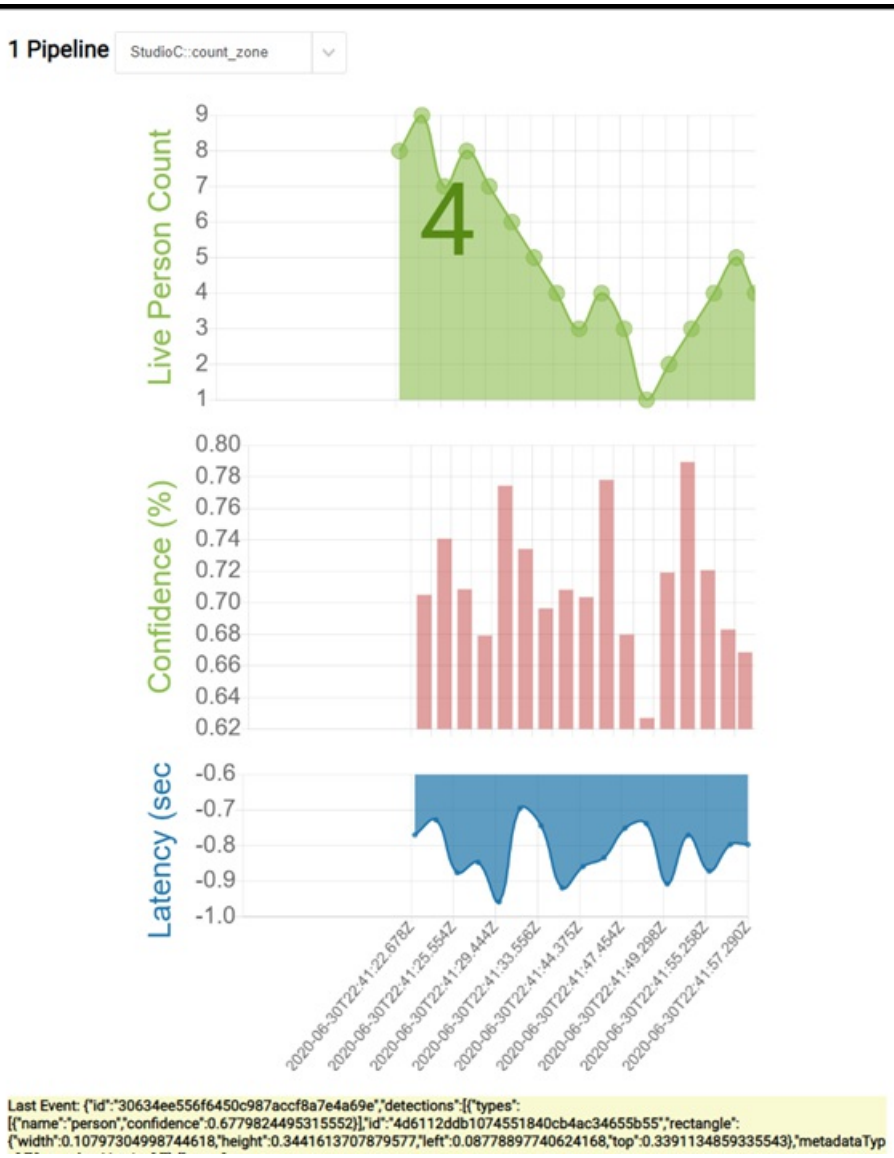
- `EventHubConsumerGroup` – The string name of the consumer group from your Azure IoT Hub, you can create a new consumer group in your IoT Hub or use the default group.
- `IotHubConnectionString` – The connection string to your Azure IoT Hub, this can be retrieved from the keys section of your Azure IoT Hub resource



Once these 2 settings are added, click **Save**. Then click **Authentication/Authorization** in the left navigation menu, and update it with the desired level of authentication. We recommend Azure Active Directory (Azure AD) express.

Test the app

Go to the Azure Service and verify the deployment was successful, and the web app is running. Navigate to the configured url: `<yourapp>.azurewebsites.net` to view the running app.



Get the PersonCount source code

If you'd like to view or modify the source code for this application, you can find it on GitHub.

Next steps

- Configure Spatial Analysis operations
- Logging and troubleshooting
- Camera placement guide
- Zone and line placement guide

Telemetry and troubleshooting

6/8/2021 • 13 minutes to read • Edit Online

Spatial Analysis includes a set of features to monitor the health of the system and help with diagnosing issues.

Enable visualizations

To enable a visualization of AI Insights events in a video frame, you need to use the `.debug` version of a Spatial Analysis operation on a desktop machine. The visualization is not possible on Azure Stack Edge devices. There are four debug operations available.

If your device is not an Azure Stack Edge device, edit the deployment manifest file for desktop machines to use the correct value for the `DISPLAY` environment variable. It needs to match the `$DISPLAY` variable on the host computer. After updating the deployment manifest, redeploy the container.

After the deployment has completed, you might have to copy the `.Xauthority` file from the host computer to the container, and restart it. In the sample below, `peopleanalytics` is the name of the container on the host computer.

```
sudo docker cp $XAUTHORITY peopleanalytics:/root/.Xauthority
sudo docker stop peopleanalytics
sudo docker start peopleanalytics
xhost +
```

Collect system health telemetry

Telegraf is an open source image that works with Spatial Analysis, and is available in the Microsoft Container Registry. It takes the following inputs and sends them to Azure Monitor. The telegraf module can be built with desired custom inputs and outputs. The telegraf module configuration in Spatial Analysis is part of the deployment manifest (linked above). This module is optional and can be removed from the manifest if you don't need it.

Inputs:

1. Spatial Analysis Metrics
2. Disk Metrics
3. CPU Metrics
4. Docker Metrics
5. GPU Metrics

Outputs:

1. Azure Monitor

The supplied Spatial Analysis telegraf module will publish all the telemetry data emitted by the Spatial Analysis container to Azure Monitor. See the Azure Monitor for information on adding Azure Monitor to your subscription.

After setting up Azure Monitor, you will need to create credentials that enable the module to send telemetry. You can use the Azure portal to create a new Service Principal, or use the Azure CLI command below to create one.

NOTE

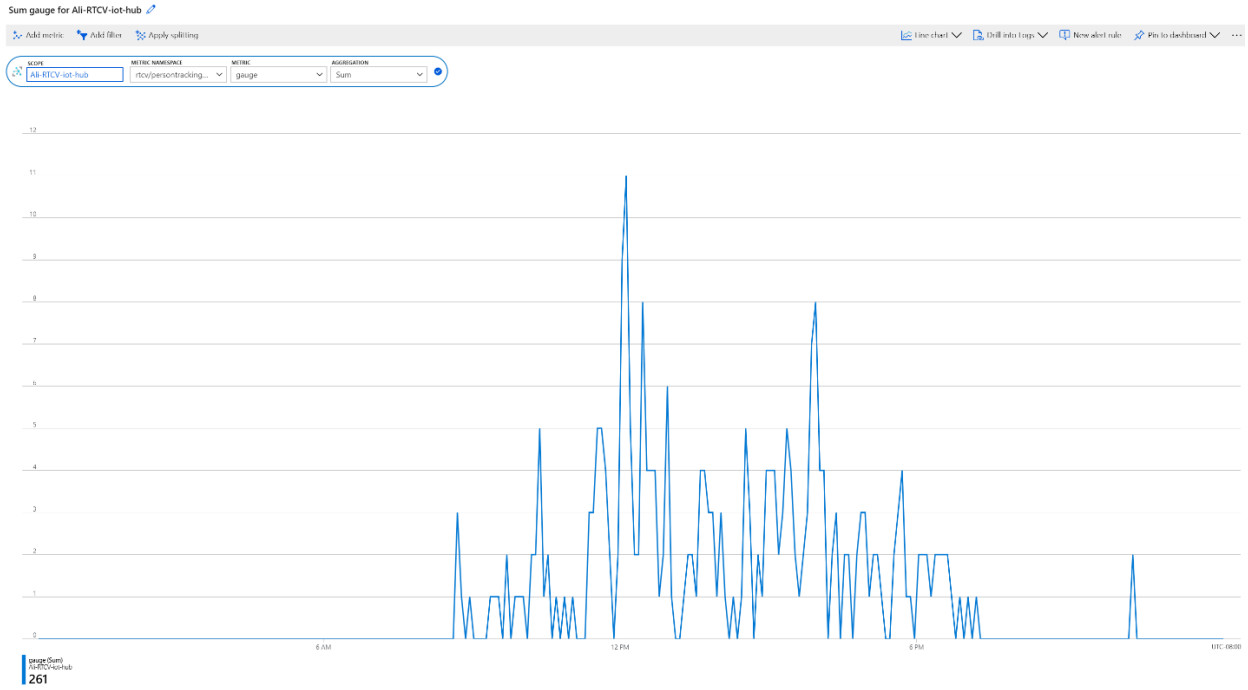
This command requires you to have Owner privileges on the subscription.

```
# Find your Azure IoT Hub resource ID by running this command. The resource ID  should start with something
like
# "/subscriptions/b60d6458-1234-4be4-9885-c7e73af9ced8/resourceGroups/..."
az iot hub list

# Create a Service Principal with `Monitoring Metrics Publisher` role in the IoTHub resource:
# Save the output from this command. The values will be used in the deployment manifest. The password won't
be shown again so make sure to write it down
az ad sp create-for-rbac --role="Monitoring Metrics Publisher" --name "<principal name>" --scopes="<resource
ID of IoT Hub>"
```

In the deployment manifest for your Azure Stack Edge device, desktop machine, or Azure VM with GPU, look for the *telegraf* module, and replace the following values with the Service Principal information from the previous step and redeploy.

```
"telegraf": {
  "settings": {
  "image":   "mcr.microsoft.com/azure-cognitive-services/vision/spatial-analysis/telegraf:1.0",
  "createOptions":   "{\"HostConfig\":{\"Runtime\":\"nvidia\",\"NetworkMode\":\"azure-iot-
edge\",\"Memory\":33554432,\"Binds\":[\"/var/run/docker.sock:/var/run/docker.sock\"]}}"
},
"type": "docker",
"env": {
    "AZURE_TENANT_ID": {
        "value": "<Tenant Id>"
    },
    "AZURE_CLIENT_ID": {
        "value": "Application Id"
    },
    "AZURE_CLIENT_SECRET": {
        "value": "<Password>"
    },
    "region": {
        "value": "<Region>"
    },
    "resource_id": {
        "value":
"/subscriptions/{subscriptionId}/resourceGroups/{resoureGroupName}/providers/Microsoft.Devices/IotHubs/{IotH
ub}"
    },
...
```

Once the telegraf module is deployed, the reported metrics can be accessed either through the Azure Monitor service, or by selecting **Monitoring** in the IoT Hub on the Azure portal.



System health events

EVENT NAME	DESCRIPTION
archon_exit	Sent when a user changes the Spatial Analysis module status from *running* to *stopped*.
archon_error	Sent when any of the processes inside the container crash. This is a critical error.
InputRate	The rate at which the graph processes video input. Reported every 5 minutes.
OutputRate	The rate at which the graph outputs AI insights. Reported every 5 minutes.
archon_allGraphsStarted	Sent when all graphs have finished starting up.
archon_configchange	Sent when a graph configuration has changed.
archon_graphCreationFailed	Sent when the graph with the reported `graphId` fails to start.
archon_graphCreationSuccess	Sent when the graph with the reported `graphId` starts successfully.
archon_graphCleanup	Sent when the graph with the reported `graphId` cleans up and exits.
archon_graphHeartbeat	Heartbeat sent every minute for every graph of a skill.
archon_apiKeyAuthFail	Sent when the Computer Vision resource key fails to authenticate the container for more than 24 hours, due to the following reasons: Out of Quota, Invalid, Offline.

EVENT NAME	DESCRIPTION
VideoIngesterHeartbeat	Sent every hour to indicate that video is streamed from the Video source, with the number of errors in that hour. Reported for each graph.
VideoIngesterState	Reports *Stopped* or *Started* for video streaming. Reported for each graph.

Troubleshooting an IoT Edge Device

You can use `iotedge` command line tool to check the status and logs of the running modules. For example:

- `iotedge list`: Reports a list of running modules. You can further check for errors with `iotedge logs edgeAgent`. If `iotedge` gets stuck, you can try restarting it with `iotedge restart edgeAgent`
- `iotedge logs <module-name>`
- `iotedge restart <module-name>` to restart a specific module

Collect log files with the diagnostics container

Spatial Analysis generates Docker debugging logs that you can use to diagnose runtime issues, or include in support tickets. The Spatial Analysis diagnostics module is available in the Microsoft Container Registry for you to download. In the manifest deployment file for your Azure Stack Edge Device, desktop machine, or Azure VM with GPU look for the *diagnostics* module.

In the "env" section add the following configuration:

```
"diagnostics": {
  "settings": {
  "image":   "mcr.microsoft.com/azure-cognitive-services/vision/spatial-analysis/diagnostics:1.0",
  "createOptions":   "{\"HostConfig\":{\"Mounts\":
[{\"Target\":\"/usr/bin/docker\",\"Source\":\"/home/data/docker\",\"Type\":\"bind\"},
{\"Target\":\"/var/run\",\"Source\":\"/run\",\"Type\":\"bind\"}],\"LogConfig\":{\"Config\":{\"max-
size\":\"500m\"}}}}"
  }
```

To optimize logs uploaded to a remote endpoint, such as Azure Blob Storage, we recommend maintaining a small file size. See the example below for the recommended Docker logs configuration.

```
{
    "HostConfig": {
        "LogConfig": {
            "Config": {
                "max-size": "500m",
                "max-file": "1000"
            }
        }
    }
}
```

Configure the log level

Log level configuration allows you to control the verbosity of the generated logs. Supported log levels are: `none`, `verbose`, `info`, `warning`, and `error`. The default log verbose level for both nodes and platform is `info`.

Log levels can be modified globally by setting the `ARCHON_LOG_LEVEL` environment variable to one of the allowed values. It can also be set through the IoT Edge Module Twin document either globally, for all deployed skills, or

for every specific skill by setting the values for `platformLogLevel` and `nodesLogLevel` as shown below.

```
{
    "version": 1,
    "properties": {
        "desired": {
            "globalSettings": {
                "platformLogLevel": "verbose"
            },
            "graphs": {
                "samplegraph": {
                    "nodesLogLevel": "verbose",
                    "platformLogLevel": "verbose"
                }
            }
        }
    }
}
```

Collecting Logs

> **NOTE**
> The `diagnostics` module does not affect the logging content, it is only assists in collecting, filtering, and uploading existing logs. You must have Docker API version 1.40 or higher to use this module.

The sample deployment manifest file for your Azure Stack Edge device, desktop machine, or Azure VM with GPU includes a module named `diagnostics` that collects and uploads logs. This module is disabled by default and should be enabled through the IoT Edge module configuration when you need to access logs.

The `diagnostics` collection is on-demand and controlled via an IoT Edge direct method, and can send logs to an Azure Blob Storage.

Configure diagnostics upload targets

From the IoT Edge portal, select your device and then the **diagnostics** module. In the sample Deployment manifest file for your Azure Stack Edge device, desktop machines, or Azure VM with GPU look for the **Environment Variables** section for diagnostics, named `env`, and add the following information:

Configure Upload to Azure Blob Storage

1. Create your own Azure Blob Storage account, if you haven't already.
2. Get the **Connection String** for your storage account from the Azure portal. It will be located in **Access Keys**.
3. Spatial Analysis logs will be automatically uploaded into a Blob Storage container named *rtcvlogs* with the following file name format: `{CONTAINER_NAME}/{START_TIME}-{END_TIME}-{QUERY_TIME}.log`.

```
"env":{
    "IOTEDGE_WORKLOADURI":"fd://iotedge.socket",
    "AZURE_STORAGE_CONNECTION_STRING":"XXXXXX",   //from the Azure Blob Storage account
    "ARCHON_LOG_LEVEL":"info"
}
```

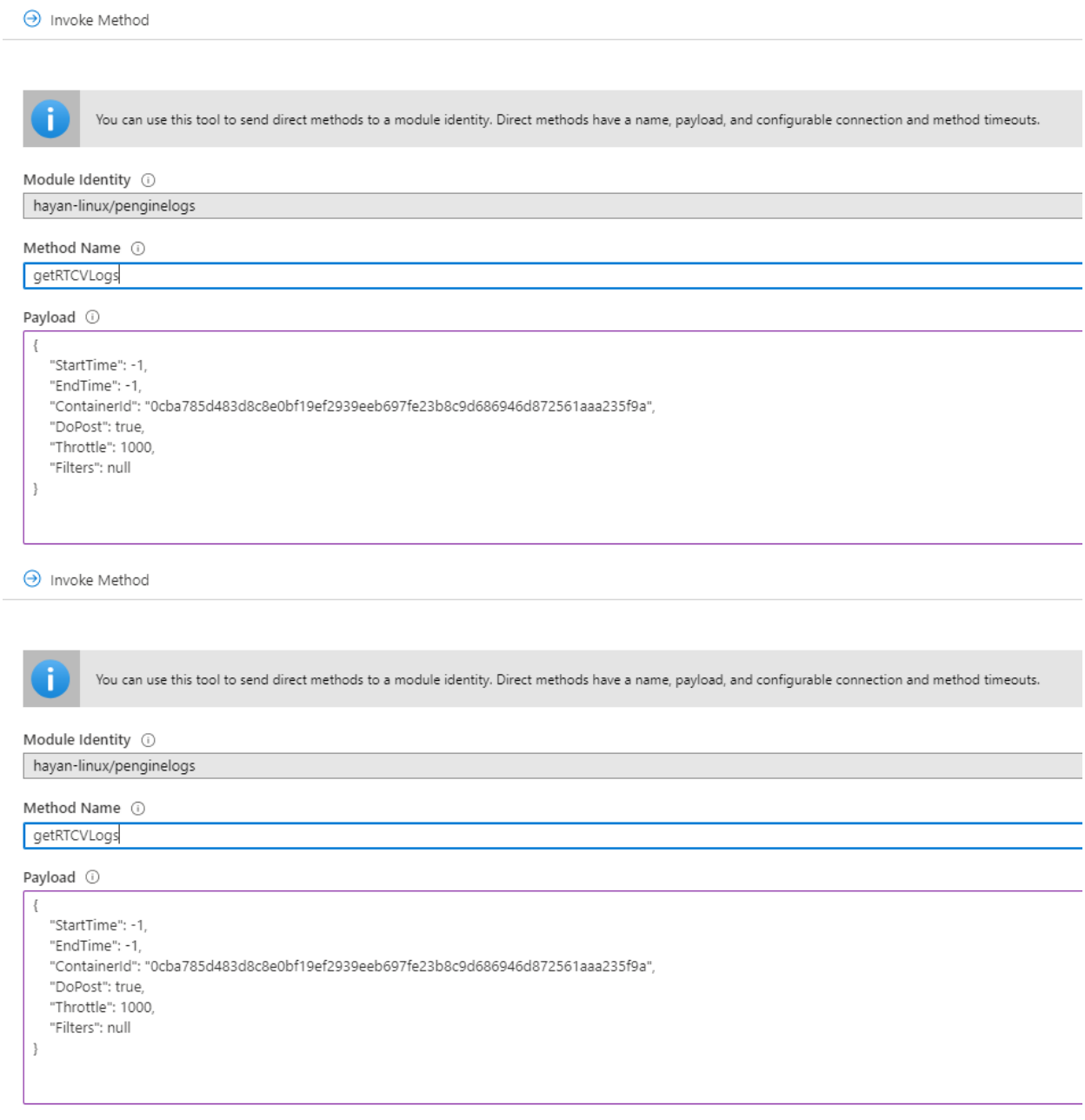
Uploading Spatial Analysis logs

Logs are uploaded on-demand with the `getRTCVLogs` IoT Edge method, in the `diagnostics` module.

1. Go to your IoT Hub portal page, select **Edge Devices**, then select your device and your diagnostics module.
2. Go to the details page of the module and click on the ***direct method*** tab.

3. Type `getRTCVLogs` on Method Name, and a json format string in payload. You can enter `{}`, which is an empty payload.
4. Set the connection and method timeouts, and click **Invoke Method**.
5. Select your target container, and build a payload json string using the parameters described in the **Logging syntax** section. Click **Invoke Method** to perform the request.

> **NOTE**
> Invoking the `getRTCVLogs` method with an empty payload will return a list of all containers deployed on the device. The method name is case sensitive. You will get a 501 error if an incorrect method name is given.



Logging syntax

The below table lists the parameters you can use when querying logs.

KEYWORD	DESCRIPTION	DEFAULT VALUE

KEYWORD	DESCRIPTION	DEFAULT VALUE
StartTime	Desired logs start time, in milliseconds UTC.	`-1`, the start of the container's runtime. When `[-1.-1]` is used as a time range, the API returns logs from the last one hour.
EndTime	Desired logs end time, in milliseconds UTC.	`-1`, the current time. When `[-1.-1]` time range is used, the api returns logs from the last one hour.
ContainerId	Target container for fetching logs.	`null`, when there is no container ID. The API returns all available containers information with IDs.
DoPost	Perform the upload operation. When this is set to `false`, it performs the requested operation and returns the upload size without performing the upload. When set to `true`, it will initiate the asynchronous upload of the selected logs	`false`, do not upload.
Throttle	Indicate how many lines of logs to upload per batch	`1000`, Use this parameter to adjust post speed.
Filters	Filters logs to be uploaded	`null`, filters can be specified as key value pairs based on the Spatial Analysis logs structure: `[UTC, LocalTime, LOGLEVEL,PID, CLASS, DATA]`. For example: `{"TimeFilter":[-1,1573255761112]}, {"TimeFilter":[-1,1573255761112]}, {"CLASS":["myNode"]`

The following table lists the attributes in the query response.

KEYWORD	DESCRIPTION
DoPost	Either *true* or *false*. Indicates if logs have been uploaded or not. When you choose not to upload logs, the api returns information ***synchronously***. When you choose to upload logs, the api returns 200, if the request is valid, and starts uploading logs ***asynchronously***.
TimeFilter	Time filter applied to the logs.
ValueFilters	Keywords filters applied to the logs.
TimeStamp	Method execution start time.
ContainerId	Target container ID.
FetchCounter	Total number of log lines.

KEYWORD	DESCRIPTION
FetchSizeInByte	Total amount of log data in bytes.
MatchCounter	Valid number of log lines.
MatchSizeInByte	Valid amount of log data in bytes.
FilterCount	Total number of log lines after applying filter.
FilterSizeInByte	Total amount of log data in bytes after applying filter.
FetchLogsDurationInMiliSec	Fetch operation duration.
PaseLogsDurationInMiliSec	Filter operation duration.
PostLogsDurationInMiliSec	Post operation duration.

Example request

```
{
    "StartTime": -1,
    "EndTime": -1,
    "ContainerId": "5fa17e4d8056e8d16a5a998318716a77becc01b36fde25b3de9fde98a64bf29b",
    "DoPost": false,
    "Filters": null
}
```

Example response

```
{
    "status": 200,
    "payload": {
        "DoPost": false,
        "TimeFilter": [-1, 1581310339411],
        "ValueFilters": {},
        "Metas": {
            "TimeStamp": "2020-02-10T04:52:19.4365389+00:00",
            "ContainerId": "5fa17e4d8056e8d16a5a998318716a77becc01b36fde25b3de9fde98a64bf29b",
            "FetchCounter": 61,
            "FetchSizeInByte": 20470,
            "MatchCounter": 61,
            "MatchSizeInByte": 20470,
            "FilterCount": 61,
            "FilterSizeInByte": 20470,
            "FetchLogsDurationInMiliSec": 0,
            "PaseLogsDurationInMiliSec": 0,
            "PostLogsDurationInMiliSec": 0
        }
    }
}
```

Check fetch log's lines, times, and sizes, if those settings look good replace ***DoPost*** to `true` and that will push the logs with same filters to destinations.

You can export logs from the Azure Blob Storage when troubleshooting issues.

Troubleshooting the Azure Stack Edge device

The following section is provided for help with debugging and verification of the status of your Azure Stack Edge device.

Access the Kubernetes API Endpoint.

1. In the local UI of your device, go to the **Devices** page.
2. Under **Device endpoints**, copy the Kubernetes API service endpoint. This endpoint is a string in the following format: `https://compute..[device-IP-address]`.
3. Save the endpoint string. You will use this later when configuring `kubectl` to access the Kubernetes cluster.

Connect to PowerShell interface

Remotely, connect from a Windows client. After the Kubernetes cluster is created, you can manage the applications via this cluster. You will need to connect to the PowerShell interface of the device. Depending on the operating system of client, the procedures to remotely connect to the device may be different. The following steps are for a Windows client running PowerShell.

> **TIP**
> - Before you begin, make sure that your Windows client is running Windows PowerShell 5.0 or later.
> - PowerShell is also available on Linux.

1. Run a Windows PowerShell session as an Administrator.
    a. Make sure that the Windows Remote Management service is running on your client. At the command prompt, type `winrm quickconfig`.
2. Assign a variable for the device IP address. For example, `$ip = "<device-ip-address>"`.
3. Use the following command to add the IP address of your device to the client's trusted hosts list.

```
Set-Item WSMan:\localhost\Client\TrustedHosts $ip -Concatenate -Force
```

4. Start a Windows PowerShell session on the device.

```
Enter-PSSession -ComputerName $ip -Credential $ip\EdgeUser -ConfigurationName Minishell
```

5. Provide the password when prompted. Use the same password that is used to sign into the local web interface. The default local web interface password is `Password1`.

Access the Kubernetes cluster

After the Kubernetes cluster is created, you can use the `kubectl` command line tool to access the cluster.

1. Create a new namespace.

```
New-HcsKubernetesNamespace -Namespace
```

2. Create a user and get a config file. This command will output configuration information for the Kubernetes cluster. Copy this information and save it in a file named *config*. Do not save the file a file extension.

```
New-HcsKubernetesUser -UserName
```

3. Add the *config* file to the *.kube* folder in your user profile on the local machine.

4. Associate the namespace with the user you created.

```
Grant-HcsKubernetesNamespaceAccess -Namespace -UserName
```

5. Install `kubectl` on your Windows client using the following command:

```
curl https://storage.googleapis.com/kubernetesrelease/release/v1.15.2/bin/windows/amd64/kubectl.exe -
O kubectl.exe
```

6. Add a DNS entry to the hosts file on your system.

   a. Run Notepad as administrator and open the *hosts* file located at `C:\windows\system32\drivers\etc\hosts`.

   b. Create an entry in the hosts file with the device IP address and DNS domain you got from the **Device** page in the local UI. The endpoint you should use will look similar to: `https://compute.asedevice.microsoftdatabox.com/10.100.10.10`.

7. Verify you can connect to the Kubernetes pods.

```
kubectl get pods -n "iotedge"
```

To get container logs, run the following command:

```
kubectl logs <pod-name> -n <namespace> --all-containers
```

Useful commands

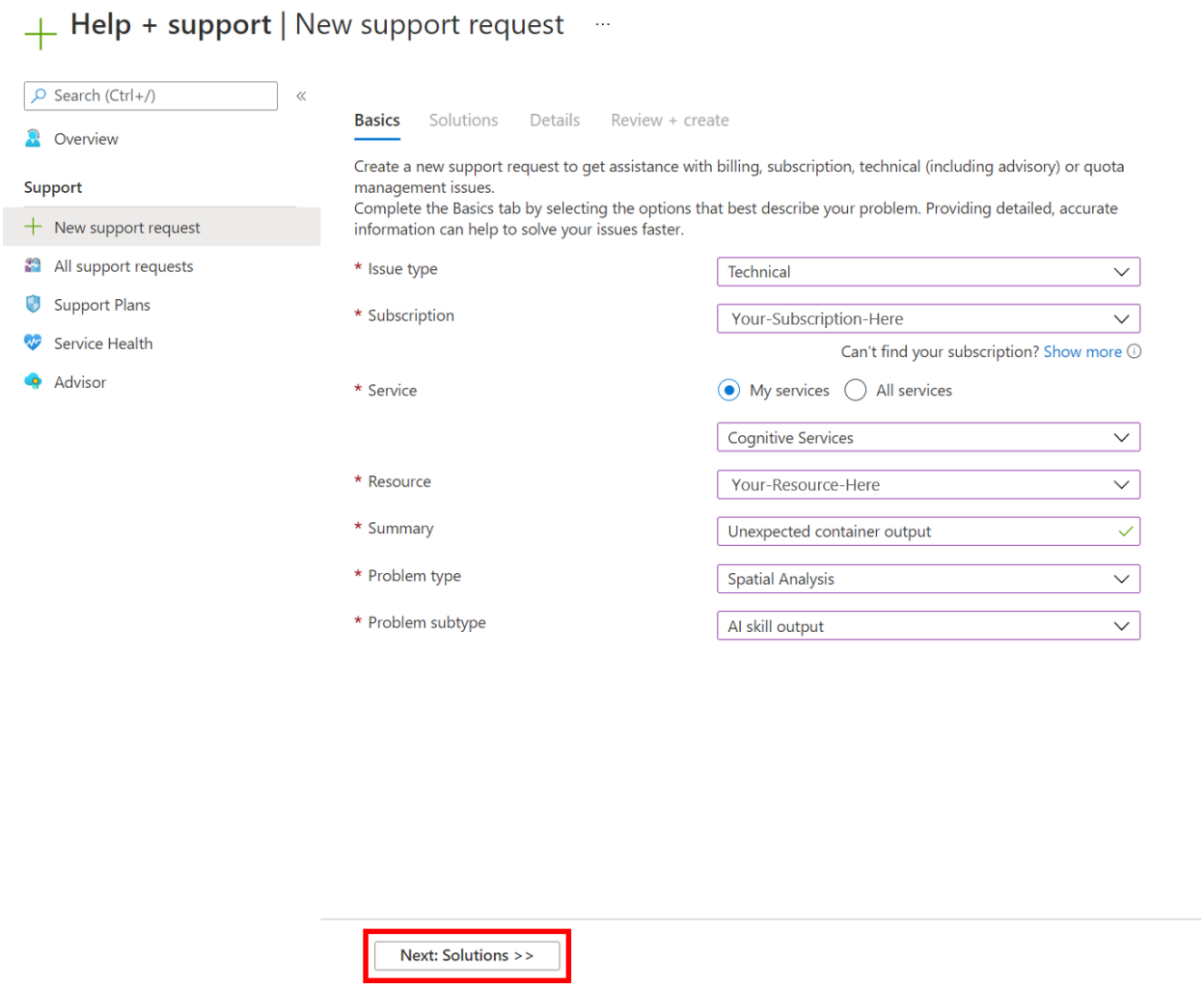
COMMAND	DESCRIPTION
`Get-HcsKubernetesUserConfig -AseUser`	Generates a Kubernetes configuration file. When using the command, copy the information into a file named *config*. Do not save the file with a file extension.
`Get-HcsApplianceInfo`	Returns information about your device.
`Enable-HcsSupportAccess`	Generates access credentials to start a support session.

How to file a support ticket for Spatial Analysis

If you need more support in finding a solution to a problem you're having with the Spatial Analysis container, follow these steps to fill out and submit a support ticket. Our team will get back to you with additional guidance.

Fill out the basics

Create a new support ticket at the New support request page. Follow the prompts to fill in the following parameters:



1. Set **Issue Type** to be `Technical`.
2. Select the subscription that you are utilizing to deploy the Spatial Analysis container.
3. Select `My services` and select `Cognitive Services` as the the service.
4. Select the resource that you are utilizing to deploy the Spatial Analysis container.
5. Write a brief description detailing the problem you are facing.
6. Select `Spatial Analysis` as your problem type.
7. Select the appropriate subtype from the drop down.
8. Select **Next: Solutions** to move on to the next page.

Recommended solutions

The next stage will offer recommended solutions for the problem type that you selected. These solutions will solve the most common problems, but if it isn't useful for your solution, select **Next: Details** to go to the next step.

Details

On this page, add some additional details about the problem you've been facing. Be sure to include as much detail as possible, as this will help our engineers better narrow down the issue. Include your preferred contact method and the severity of the issue so we can contact you appropriately, and select **Next: Review + create** to move to the next step.

Review and create

Review the details of your support request to ensure everything is accurate and represents the problem effectively. Once you are ready, select **Create** to send the ticket to our team! You will receive an email confirmation once your ticket is received, and our team will work to get back to you as soon as possible. You can view the status of your ticket in the Azure portal.

Next steps

- Deploy a People Counting web application
- Configure Spatial Analysis operations
- Camera placement guide
- Zone and line placement guide

Zone and Line Placement Guide

6/8/2021 • 2 minutes to read • Edit Online

This article provides guidelines for how to define zones and lines for Spatial Analysis operations to achieve accurate analysis of peoples movements in a space. This applies to all operations.

Zones and lines are defined using the JSON SPACEANALYSIS_CONFIG parameter. See the Spatial Analysis operations article for more information.

Guidelines for drawing zones

Remember that every space is different; you'll need to update the position or size depending on your needs.

If you want to see a specific section of your camera view, create the largest zone that you can, covering the specific floor area that you're interested in but not including other areas that you're not interested in. This increases the accuracy of the data collected and prevents false positives from areas you don't want to track. Be careful when placing the corners of your polygon and make sure they're not outside the area you want to track.

Example of a well-shaped zone

The zone should be big enough to accommodate three people standing along each edge and focused on the area of interest. Spatial Analysis will identify people whose feet are placed in the zone, so when drawing zones on the 2D image, imagine the zone as a carpet laying on the floor.

Examples of zones that aren't well-shaped

The following examples show poorly shaped zones. In these examples, the area of interest is the space in front of the *It's Game Time* display.

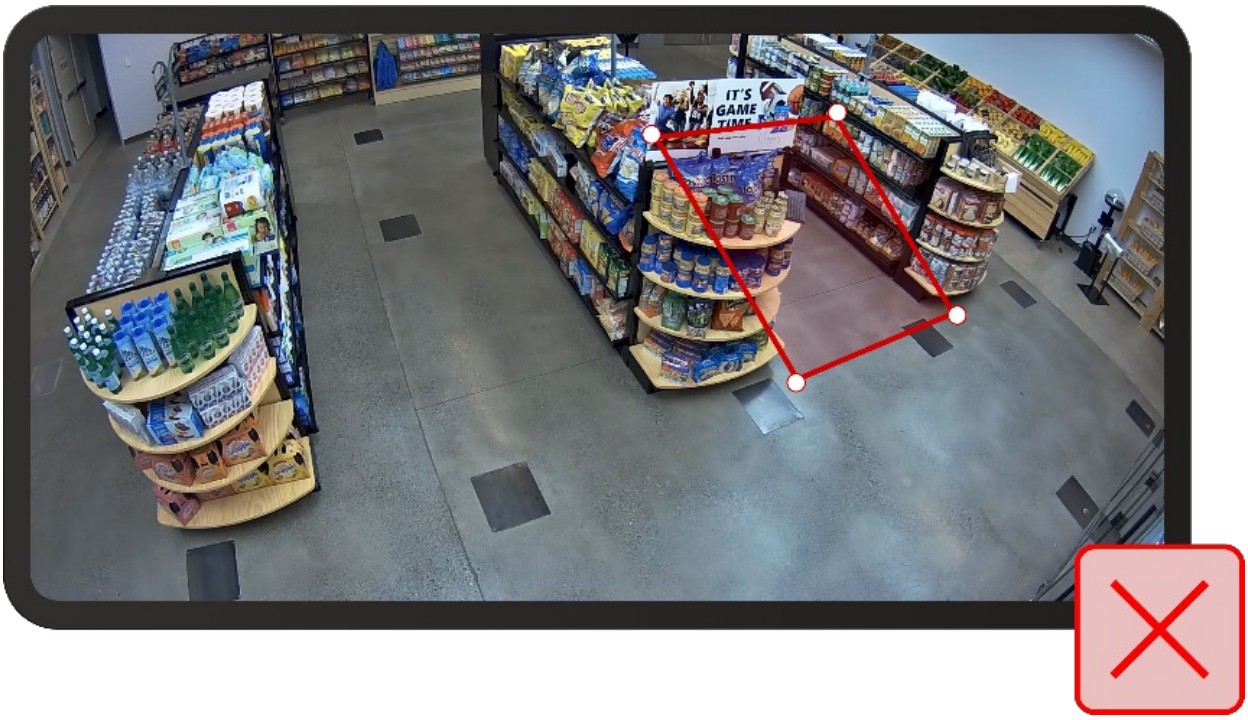
Zone is not on the floor.

Zone is too small.

Zone doesn't fully capture the area around the display.

Zone is too close to the edge of the camera image and doesn't capture the right display.

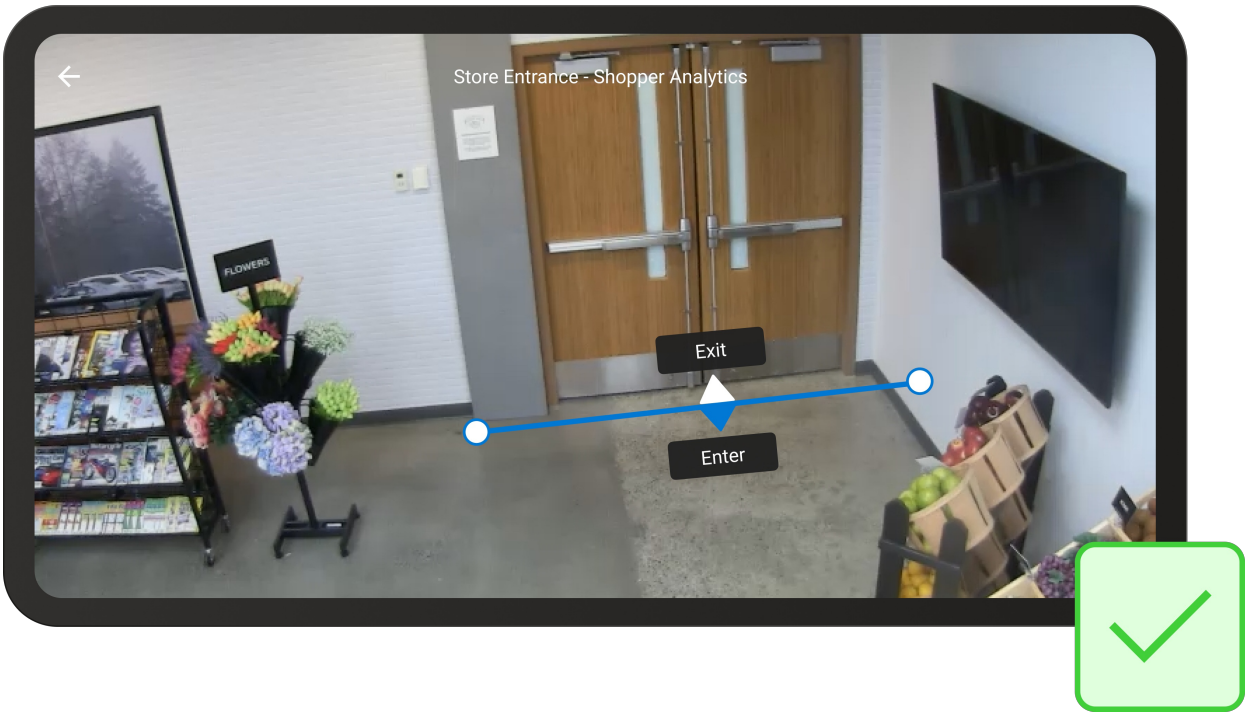
Zone is partially blocked by the shelf, so people and floor aren't fully visible.



Example of a well-shaped line

The line should be long enough to accommodate the entire entrance. Spatial Analysis will identify people whose feet cross the line, so when drawing lines on the 2D image imagine you're drawing them as if they lie on the floor.

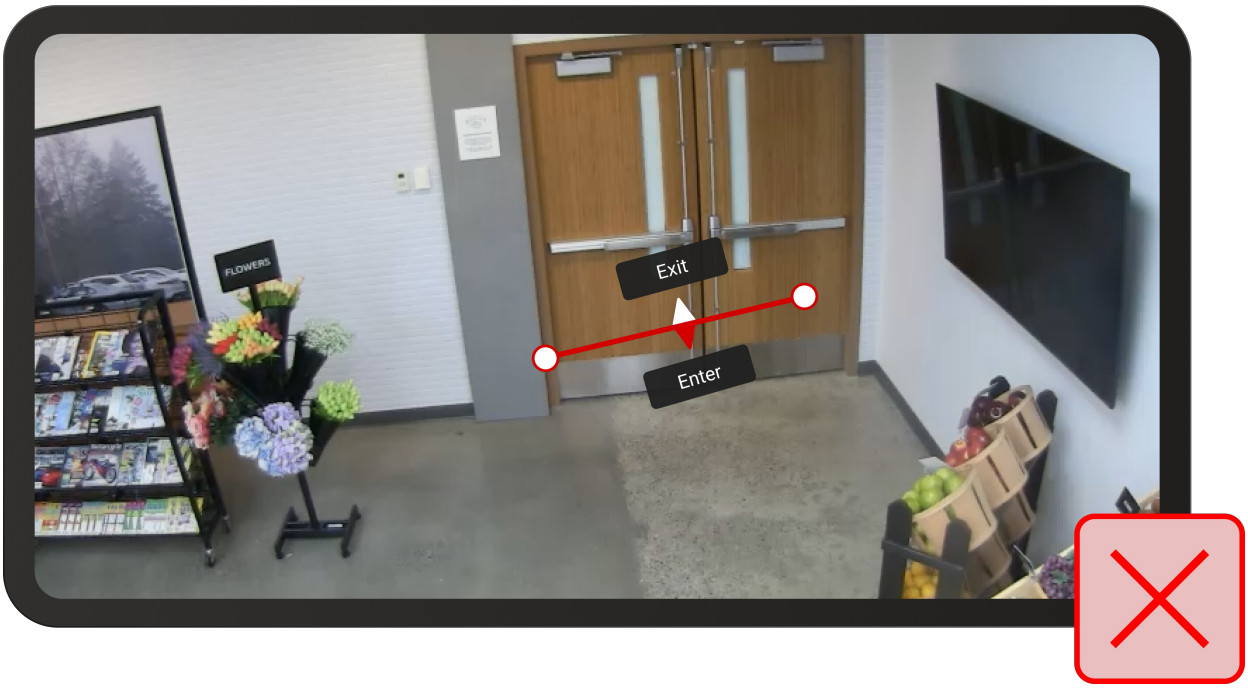
If possible, extend the line wider than the actual entrance. If this will not result in extra crossings (as in the image below when the line is against a wall) then extend it.



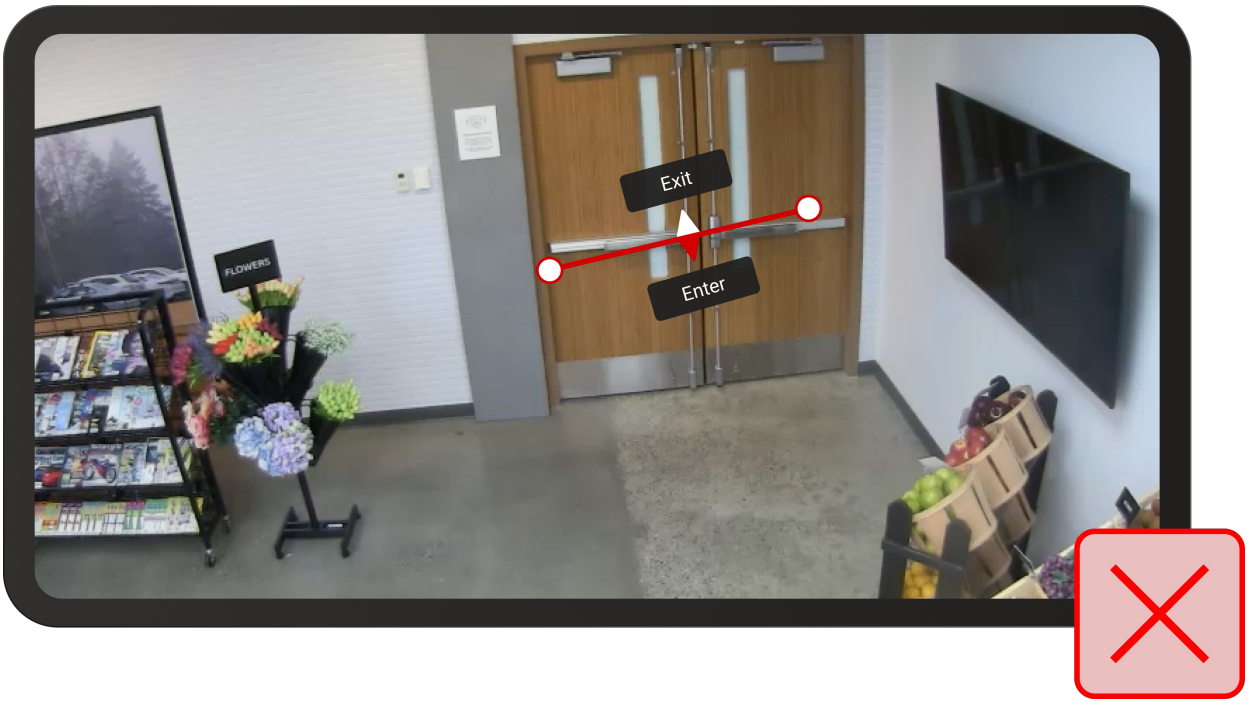
Examples of lines that aren't well-shaped

The following examples show poorly defined lines.

Line doesn't cover the entire entry way on the floor.



Line is too high and doesn't cover the entirety of the door.



Next steps

- Deploy a People Counting web application
- Configure Spatial Analysis operations
- Logging and troubleshooting
- Camera placement guide

Camera placement guide

6/8/2021 • 4 minutes to read • Edit Online

This article provides camera placement recommendations for Spatial Analysis (public preview). It includes general guidelines as well as specific recommendations for height, angle, and camera-to-focal-point-distance for all the included operations.

> **NOTE**
>
> This guide is designed for the Axis M3045-V camera. This camera will use resolution 1920x1080, 106 degree horizontal field of view, 59 degree vertical field of view and a fixed 2.8mm focal length. The principles below will apply to all cameras, but specific guidelines around camera height and camera-to-focal-point distance will need to be adjusted for use with other cameras.

General guidelines

Consider the following general guidelines when positioning cameras for Spatial Analysis:

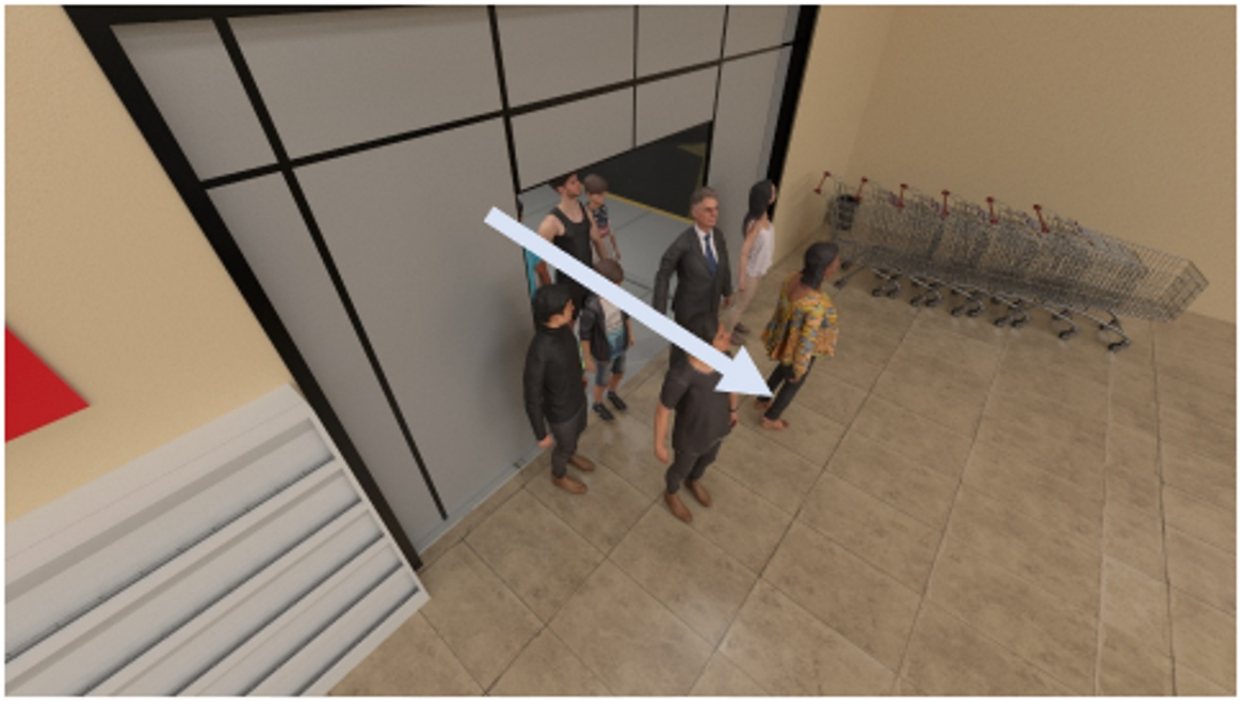
- **Lighting height.** Place cameras below lighting fixtures so the fixtures don't block the cameras.
- **Obstructions.** To avoid obstructing camera views, take note of obstructions such as poles, signage, shelving, walls, and existing LP cameras.
- **Environmental backlighting.** Outdoor backlighting affects camera image quality. To avoid severe backlighting conditions, avoid directing cameras at external-facing windows and glass doors.
- **Local privacy rules and regulations.** Local regulations may restrict what cameras can capture. Make sure that you understand local rules and regulations before placing cameras.
- **Building structure.** HVAC, sprinklers, and existing wiring may limit hard mounting of cameras.
- **Cable management.** Make sure you can route an ethernet cable from planned camera mounting locations to the Power Over Internet (PoE) switch.

Height, focal-point distance, and angle

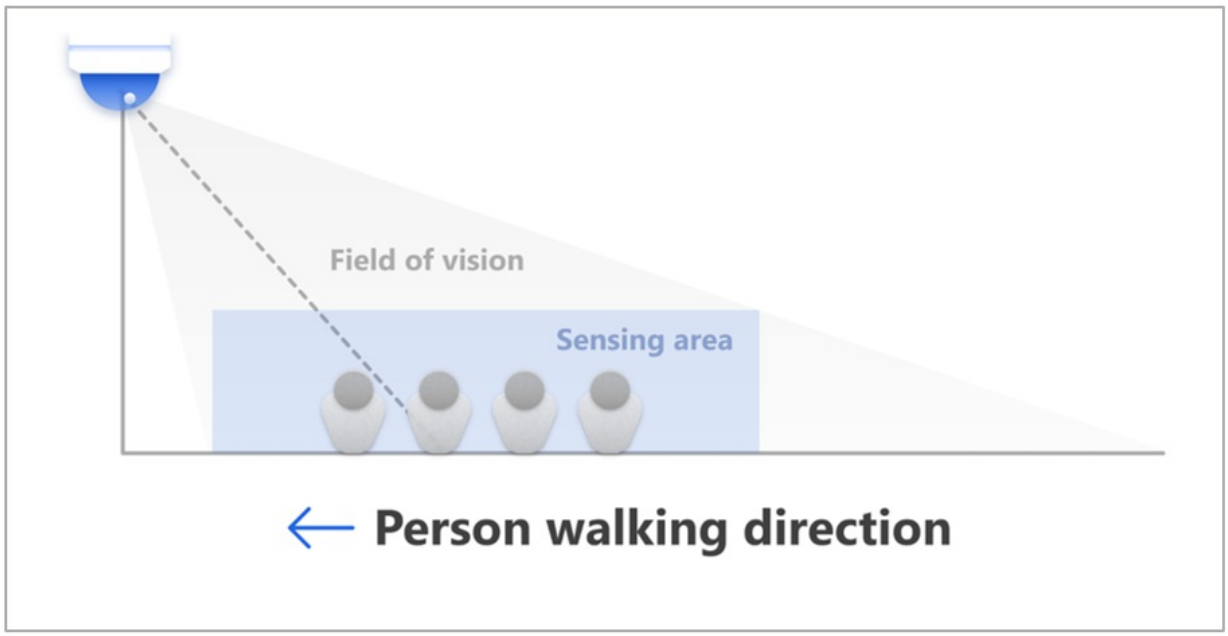
You need to consider three things when deciding how to install a camera for Spatial Analysis:

- Camera height
- Camera-to-focal-point distance
- The angle of the camera relative to the floor plane

It's also important to know the direction that the majority of people walk (person walking direction) in relation to the camera field of view if possible. This direction is important for system performance.



The following illustration shows the elevation view for person walking direction.



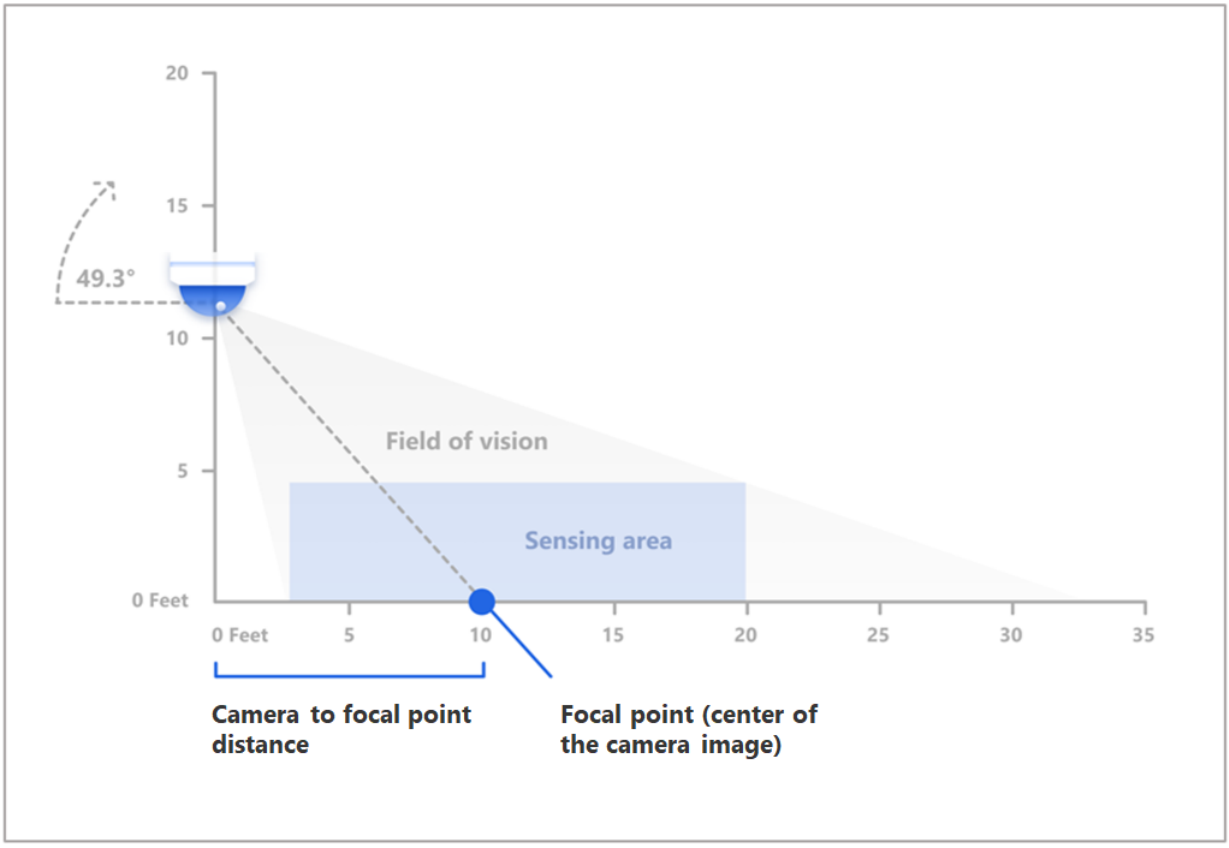
Camera height

Generally, cameras should be mounted 12-14 feet from the ground. For Face mask detection, we recommend cameras to be mounted 8-12 feet from the ground. When planning your camera mounting in this range, consider obstructions (for example: shelving, hanging lights, hanging signage, and displays) that might affect the camera view, and then adjust the height as necessary.

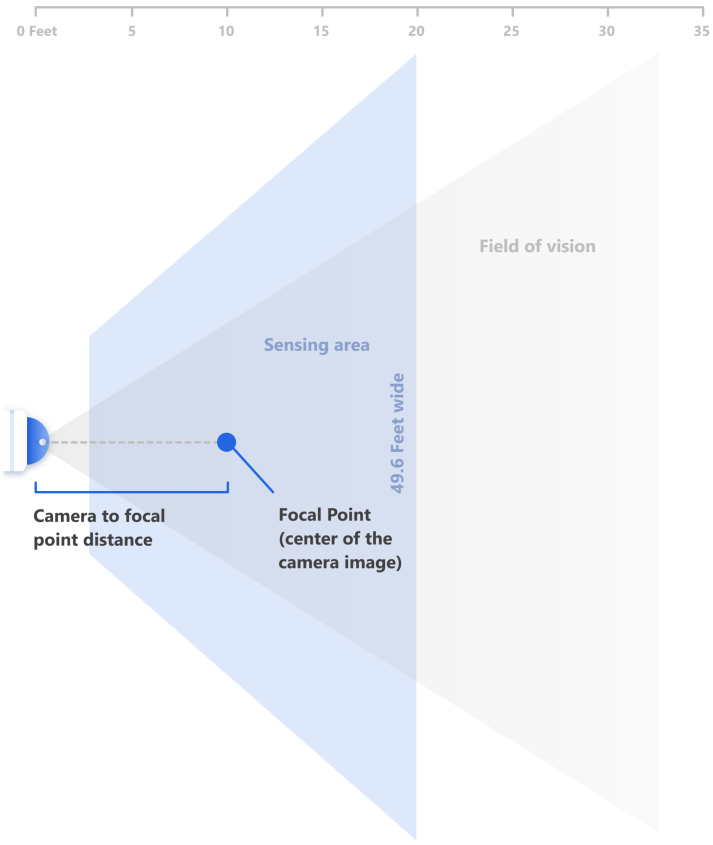
Camera-to-focal-point distance

Camera-to-focal-point distance is the linear distance from the focal point (or center of the camera image) to the camera measured on the ground.

This distance is measured on the floor plane.



From above, it looks like this:



Use the table below to determine the camera's distance from the focal point based on specific mounting heights. These distances are for optimal placement. Note that the table provides guidance below the 12'-14' recommendation since some ceilings can limit height. For Face mask detection, recommended camera-to-focal-point distance (min/max) is 4′-10′ for camera height between 8′ to 12′.

CAMERA HEIGHT	CAMERA-TO-FOCAL-POINT DISTANCE (MIN/MAX)
8'	4.6'-8'
10'	5.8'-10'
12'	7'-12'
14'	8'-14"
16'	9.2'-16'
20'	11.5'-20'

The following illustration simulates camera views from the closest and farthest camera-to-focal-point distances.

CLOSEST	FARTHEST
	

Camera angle mounting ranges

This section describes acceptable camera angle mounting ranges. These mounting ranges show the acceptable range for optimal placement.

Line configuration

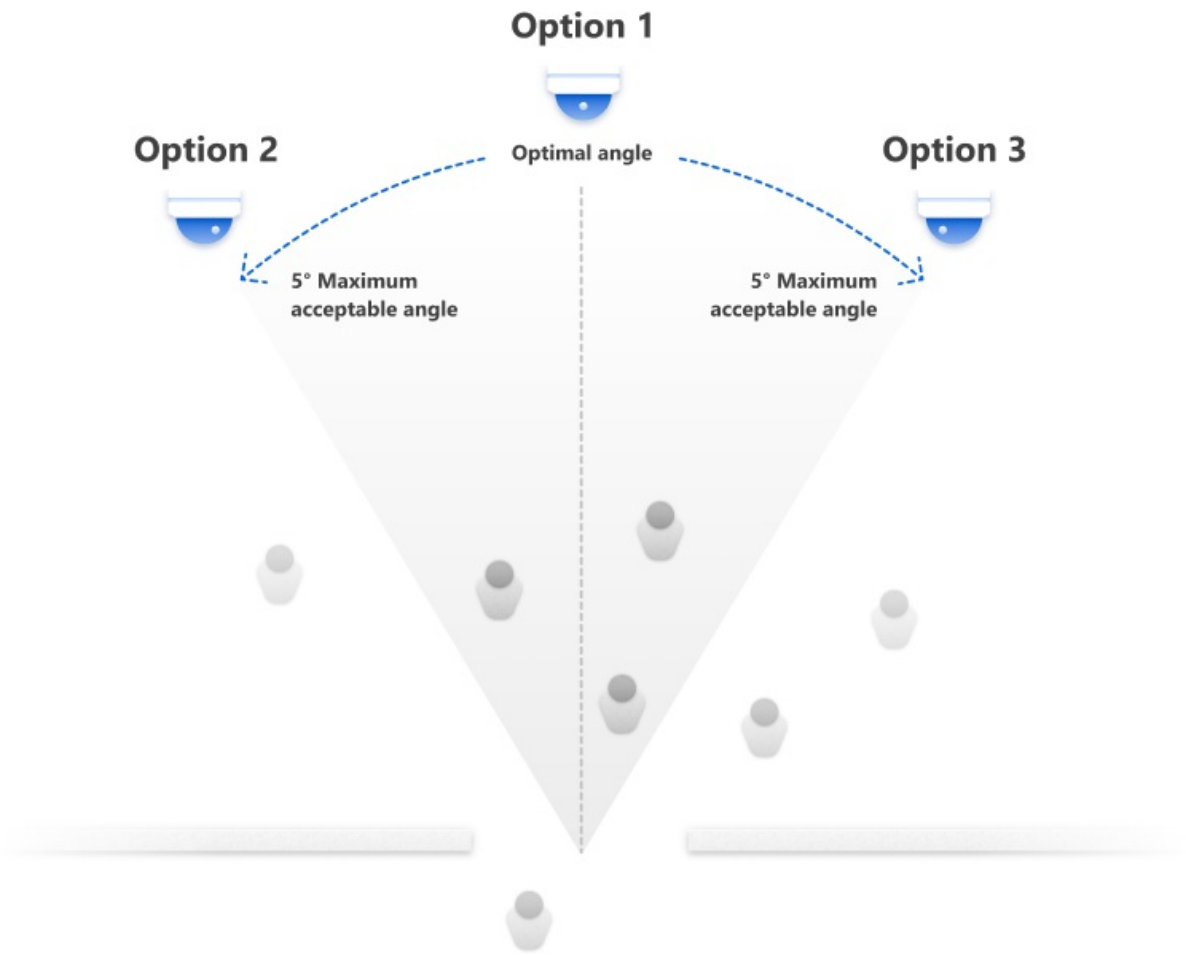
For the **cognitiveservices.vision.spatialanalysis-personcrossingline** operation, +/-5° is the optimal camera mounting angle to maximize accuracy.

For Face mask detection, +/-30 degrees is the optimal camera mounting angle for camera height between 8′ to 12′.

The following illustration simulates camera views using the leftmost (-) and rightmost (+) mounting angle recommendations for using **cognitiveservices.vision.spatialanalysis-personcrossingline** to do entrance counting in a door way.

LEFTMOST VIEW	RIGHTMOST VIEW
	

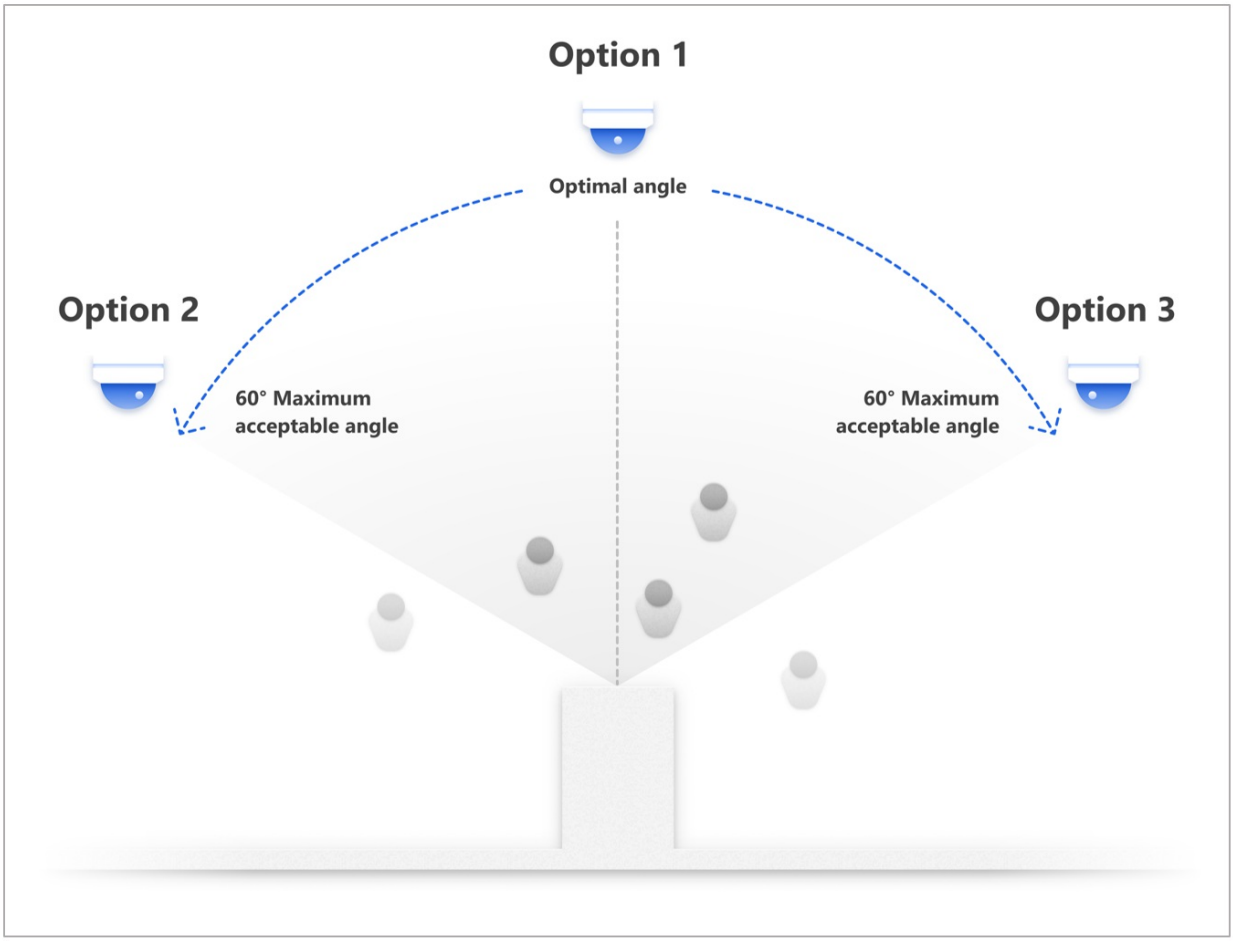
The following illustration shows camera placement and mounting angles from a birds-eye view.



Zone configuration

We recommend that you place cameras at 10 feet or more above ground to guarantee the covered area is big enough.

When the zone is next to an obstacle like a wall or shelf, mount cameras in the specified distance from the target within the acceptable 120-degree angle range as shown in the following illustration.

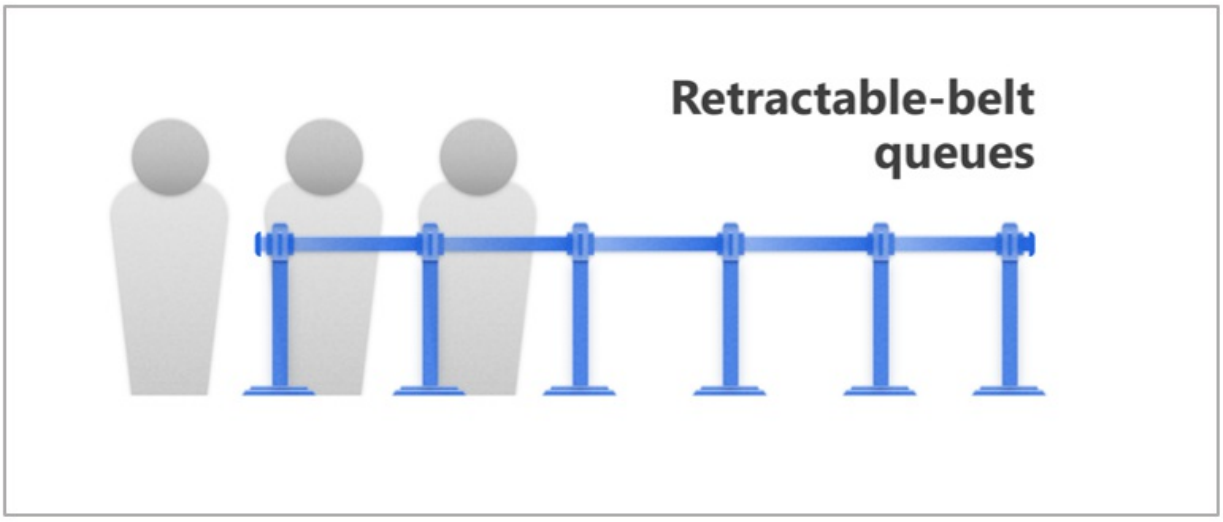


The following illustration provides simulations for the left and right camera views of an area next to a shelf.

LEFT VIEW	RIGHT VIEW
	

Queues

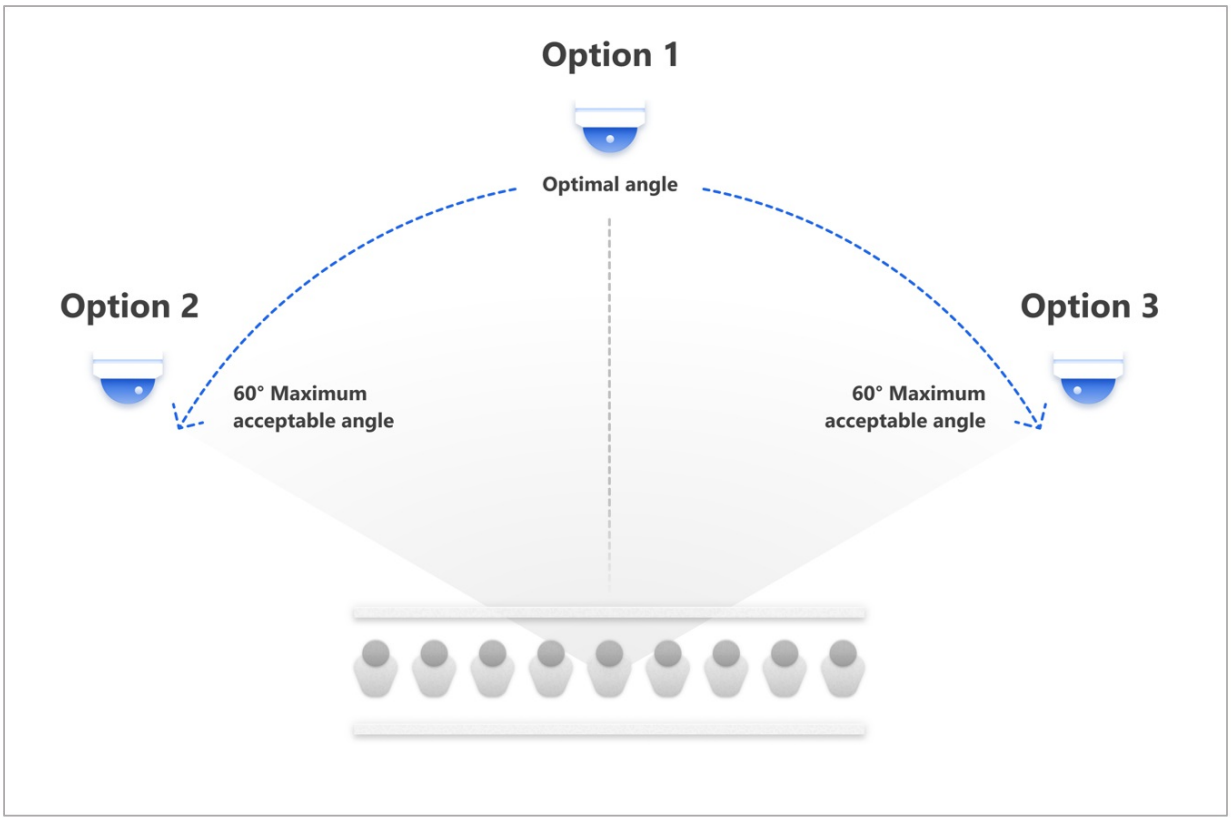
The **cognitiveservices.vision.spatialanalysis-personcount**, **cognitiveservices.vision.spatialanalysis-persondistance**, and **cognitiveservices.vision.spatialanalysis-personcrossingpolygon** skills may be used to monitor queues. For optimal queue data quality, retractable belt barriers are preferred to minimize occlusion of the people in the queue and ensure the queues location is consistent over time.



This type of barrier is preferred over opaque barriers for queue formation to maximize the accuracy of the insights from the system.

There are two types of queues: linear and zig-zag.

The following illustration shows recommendations for linear queues:



The following illustration provides simulations for the left and right camera views of linear queues. Note that you can mount the camera on the opposite side of the queue.

LEFT VIEW	RIGHT VIEW
	

For zig-zag queues, it's best to avoid placing the camera directly facing the queue line direction, as shown in the following illustration. Note that each of the four example camera positions in the illustration provide the ideal view with an acceptable deviation of +/- 15 degrees in each direction.

The following illustrations simulate the view from a camera placed in the ideal locations for a zig-zag queue.

VIEW 1	VIEW 2
	

VIEW 3	VIEW 4
	

Organic queues

Organic queue lines form organically. This style of queue is acceptable if queues don't form beyond 2-3 people and the line forms within the zone definition. If the queue length is typically more than 2-3 people, we recommend using a retractable belt barrier to help guide the queue direction and ensure the line forms within the zone definition.

Next steps

- Deploy a People Counting web application
- Configure Spatial Analysis operations

- Logging and troubleshooting
- Zone and line placement guide

Configure Azure Cognitive Services virtual networks

6/15/2021 • 16 minutes to read • Edit Online

Azure Cognitive Services provides a layered security model. This model enables you to secure your Cognitive Services accounts to a specific subset of networks. When network rules are configured, only applications requesting data over the specified set of networks can access the account. You can limit access to your resources with request filtering. Allowing only requests originating from specified IP addresses, IP ranges or from a list of subnets in Azure Virtual Networks.

An application that accesses a Cognitive Services resource when network rules are in effect requires authorization. Authorization is supported with Azure Active Directory (Azure AD) credentials or with a valid API key.

IMPORTANT

Turning on firewall rules for your Cognitive Services account blocks incoming requests for data by default. In order to allow requests through, one of the following conditions needs to be met:

- The request should originate from a service operating within an Azure Virtual Network (VNet) on the allowed subnet list of the target Cognitive Services account. The endpoint in requests originated from VNet needs to be set as the custom subdomain of your Cognitive Services account.
- Or the request should originate from an allowed list of IP addresses.

Requests that are blocked include those from other Azure services, from the Azure portal, from logging and metrics services, and so on.

NOTE

This article has been updated to use the Azure Az PowerShell module. The Az PowerShell module is the recommended PowerShell module for interacting with Azure. To get started with the Az PowerShell module, see Install Azure PowerShell. To learn how to migrate to the Az PowerShell module, see Migrate Azure PowerShell from AzureRM to Az.

Scenarios

To secure your Cognitive Services resource, you should first configure a rule to deny access to traffic from all networks (including internet traffic) by default. Then, you should configure rules that grant access to traffic from specific VNets. This configuration enables you to build a secure network boundary for your applications. You can also configure rules to grant access to traffic from select public internet IP address ranges, enabling connections from specific internet or on-premises clients.

Network rules are enforced on all network protocols to Azure Cognitive Services, including REST and WebSocket. To access data using tools such as the Azure test consoles, explicit network rules must be configured. You can apply network rules to existing Cognitive Services resources, or when you create new Cognitive Services resources. Once network rules are applied, they're enforced for all requests.

Supported regions and service offerings

Virtual networks (VNETs) are supported in regions where Cognitive Services are available. Currently multi-service resource does not support VNET. Cognitive Services supports service tags for network rules

configuration. The services listed below are included in the **CognitiveServicesManagement** service tag.

- Anomaly Detector
- Computer Vision
- Content Moderator
- Custom Vision
- Face
- Form Recognizer
- Immersive Reader
- Language Understanding (LUIS)
- Personalizer
- Speech Services
- Text Analytics
- QnA Maker
- Translator Text

> **NOTE**
> If you're using LUIS or Speech Services, the **CognitiveServicesManagement** tag only enables you use the service using the SDK or REST API. To access and use LUIS portal and/or Speech Studio from a virtual network, you will need to use the following tags:
>
> - **AzureActiveDirectory**
> - **AzureFrontDoor.Frontend**
> - **AzureResourceManager**
> - **CognitiveServicesManagement**

Change the default network access rule

By default, Cognitive Services resources accept connections from clients on any network. To limit access to selected networks, you must first change the default action.

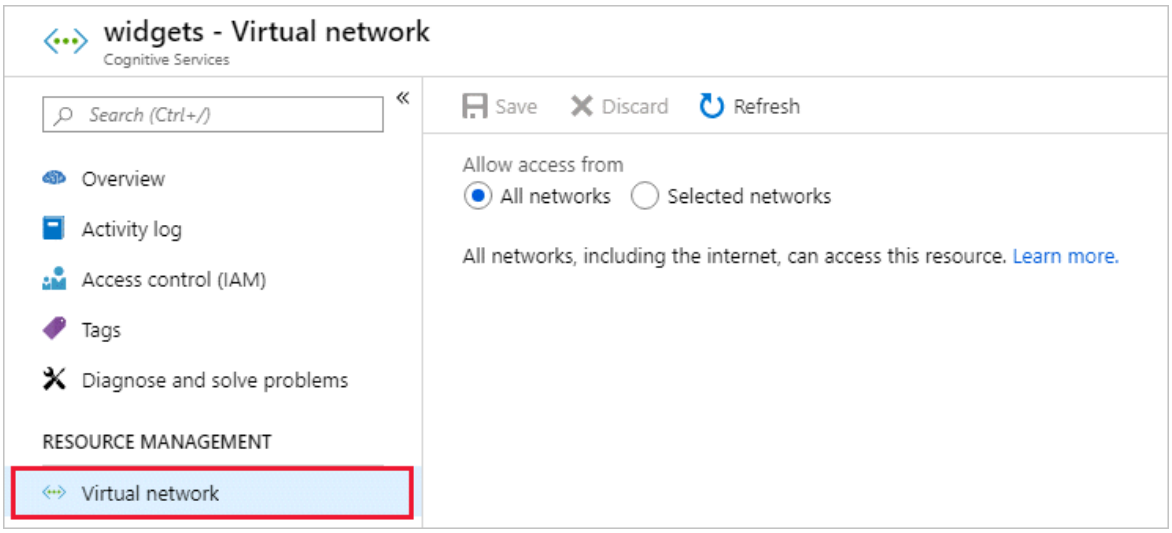
> **WARNING**
> Making changes to network rules can impact your applications' ability to connect to Azure Cognitive Services. Setting the default network rule to **deny** blocks all access to the data unless specific network rules that **grant** access are also applied. Be sure to grant access to any allowed networks using network rules before you change the default rule to deny access. If you are allow listing IP addresses for your on-premises network, be sure to add all possible outgoing public IP addresses from your on-premises network.

Managing default network access rules

You can manage default network access rules for Cognitive Services resources through the Azure portal, PowerShell, or the Azure CLI.

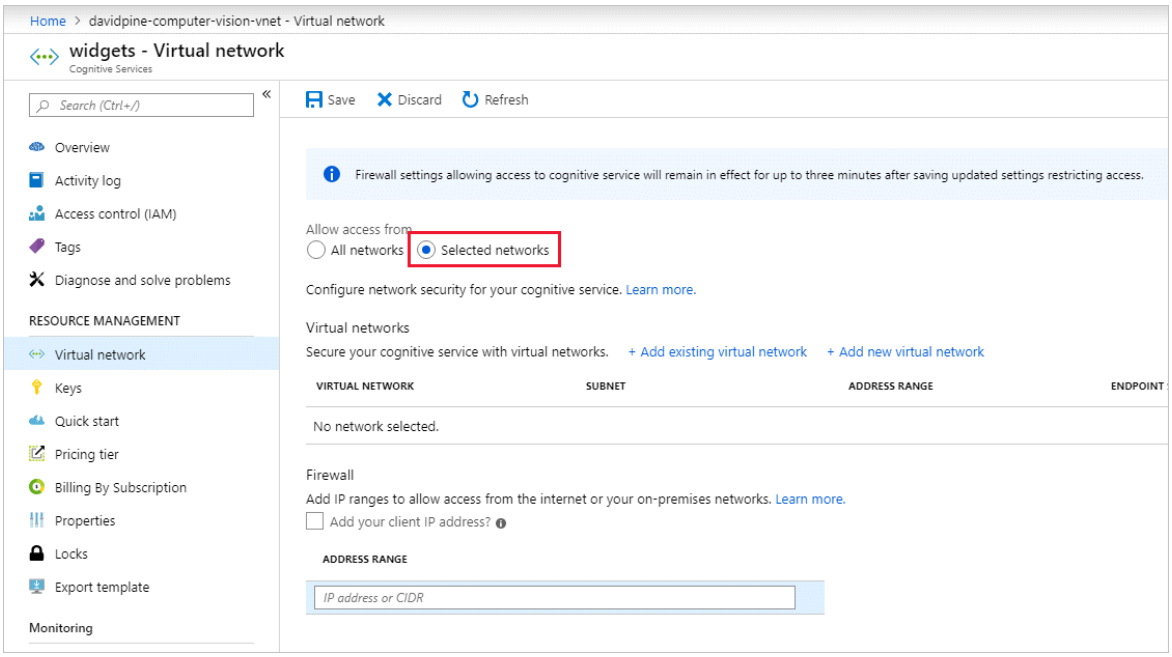
- Azure portal
- PowerShell
- Azure CLI

1. Go to the Cognitive Services resource you want to secure.
2. Select the **RESOURCE MANAGEMENT** menu called **Virtual network**.



3. To deny access by default, choose to allow access from **Selected networks**. With the **Selected networks** setting alone, unaccompanied by configured **Virtual networks** or **Address ranges** - all access is effectively denied. When all access is denied, requests attempting to consume the Cognitive Services resource aren't permitted. The Azure portal, Azure PowerShell or, Azure CLI can still be used to configure the Cognitive Services resource.

4. To allow traffic from all networks, choose to allow access from **All networks**.



5. Select **Save** to apply your changes.

Grant access from a virtual network

You can configure Cognitive Services resources to allow access only from specific subnets. The allowed subnets may belong to a VNet in the same subscription, or in a different subscription, including subscriptions belonging to a different Azure Active Directory tenant.

Enable a service endpoint for Azure Cognitive Services within the VNet. The service endpoint routes traffic from the VNet through an optimal path to the Azure Cognitive Services service. The identities of the subnet and the virtual network are also transmitted with each request. Administrators can then configure network rules for the Cognitive Services resource that allow requests to be received from specific subnets in a VNet. Clients granted access via these network rules must continue to meet the authorization requirements of the Cognitive Services resource to access the data.

Each Cognitive Services resource supports up to 100 virtual network rules, which may be combined with IP network rules.

Required permissions

To apply a virtual network rule to a Cognitive Services resource, the user must have the appropriate permissions for the subnets being added. The required permission is the default *Contributor* role, or the *Cognitive Services Contributor* role. Required permissions can also be added to custom role definitions.

Cognitive Services resource and the virtual networks granted access may be in different subscriptions, including subscriptions that are a part of a different Azure AD tenant.

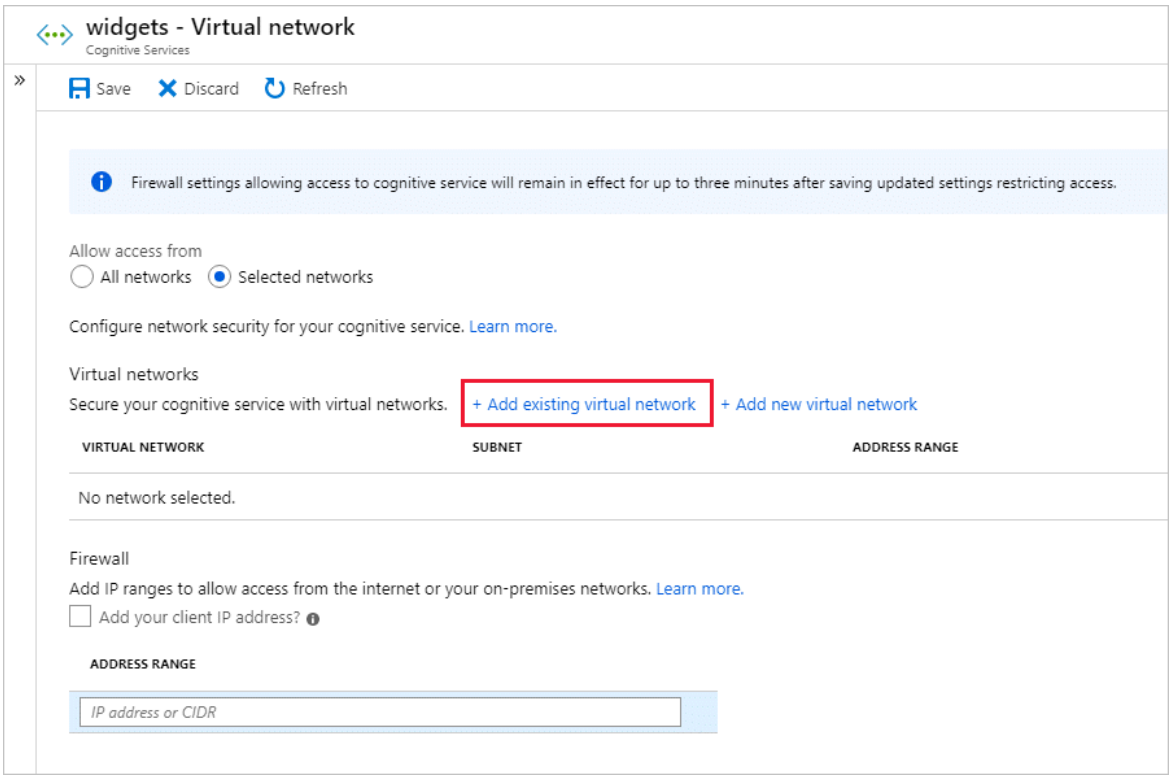
> **NOTE**
>
> Configuration of rules that grant access to subnets in virtual networks that are a part of a different Azure Active Directory tenant are currently only supported through Powershell, CLI and REST APIs. Such rules cannot be configured through the Azure portal, though they may be viewed in the portal.

Managing virtual network rules

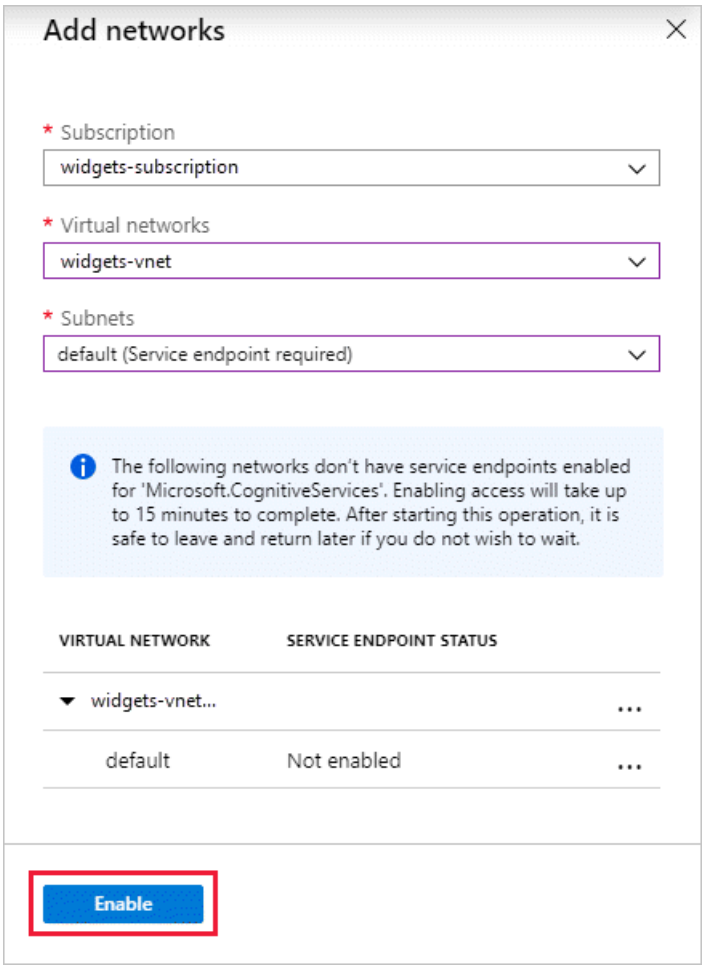
You can manage virtual network rules for Cognitive Services resources through the Azure portal, PowerShell, or the Azure CLI.

- Azure portal
- PowerShell
- Azure CLI

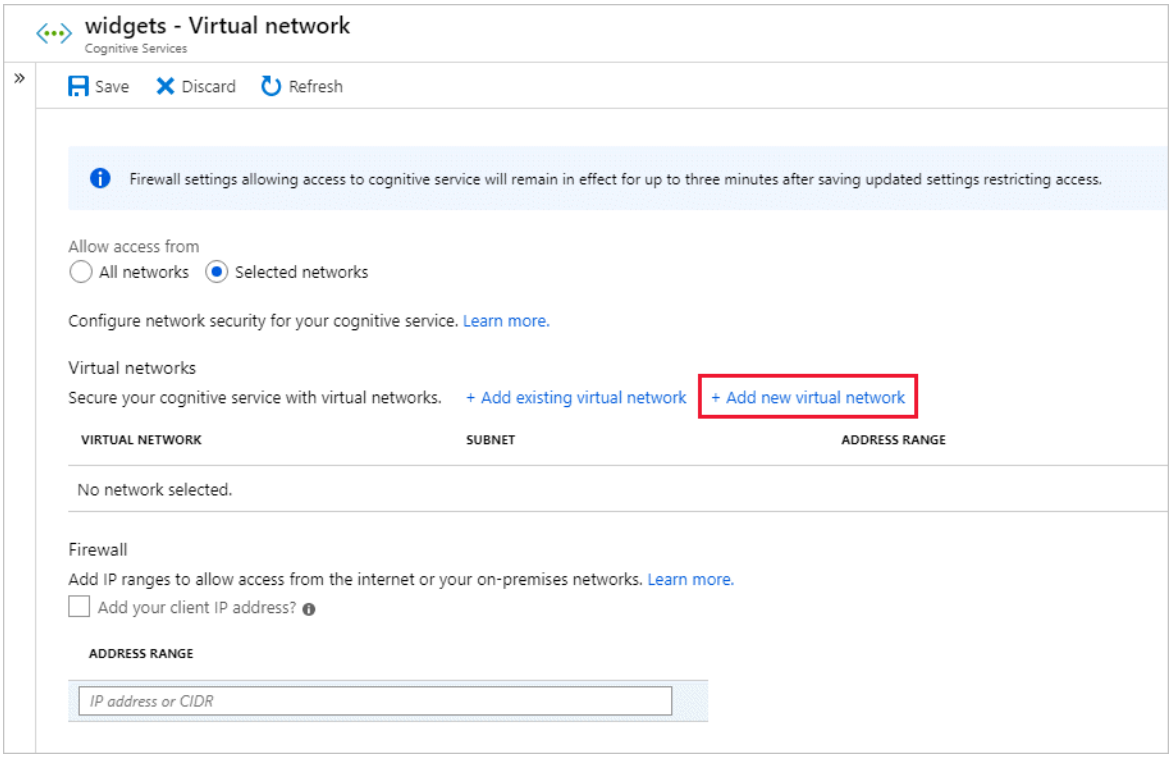
1. Go to the Cognitive Services resource you want to secure.
2. Select the **RESOURCE MANAGEMENT** menu called **Virtual network**.
3. Check that you've selected to allow access from **Selected networks**.
4. To grant access to a virtual network with an existing network rule, under **Virtual networks**, select **Add existing virtual network**.



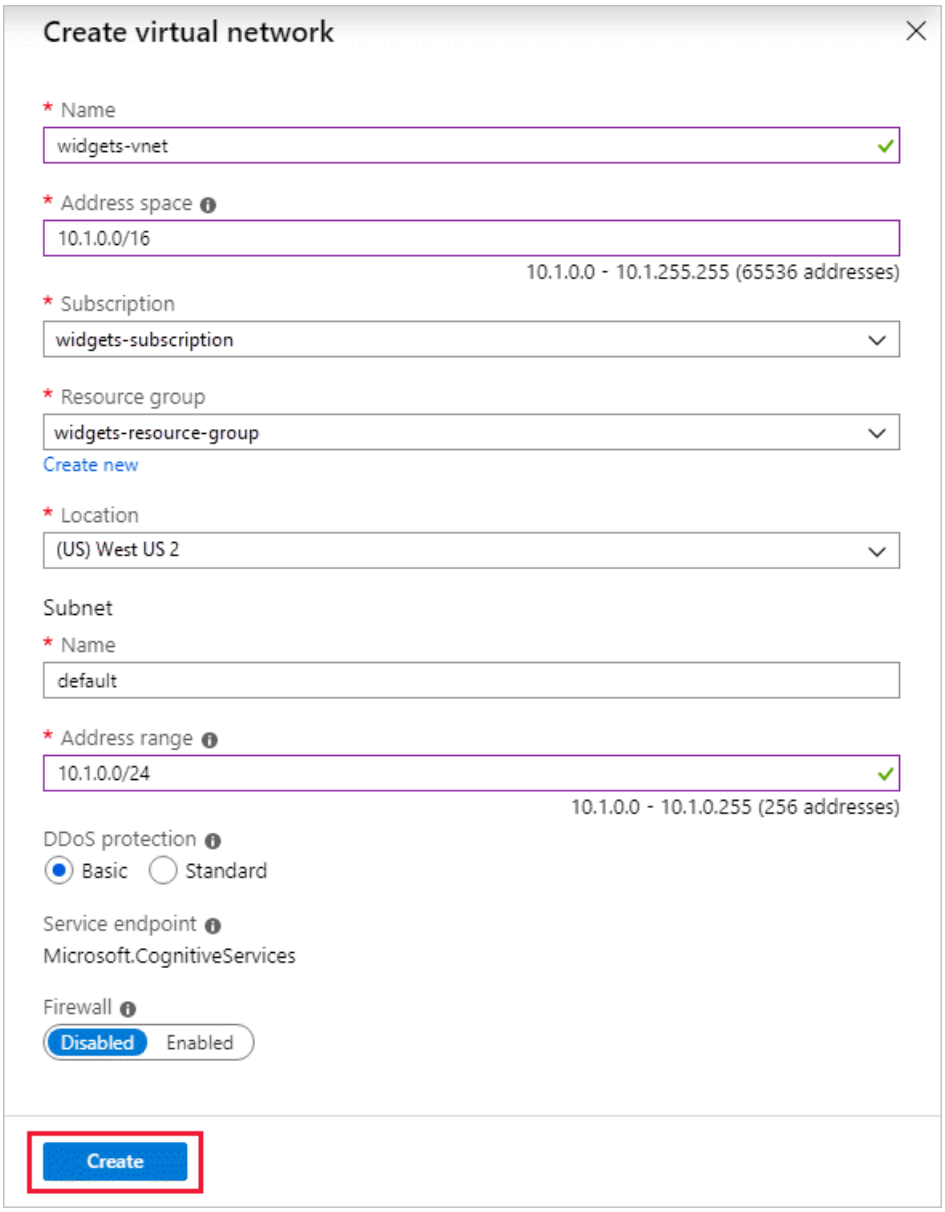
5. Select the **Virtual networks** and **Subnets** options, and then select **Enable**.



6. To create a new virtual network and grant it access, select **Add new virtual network**.



7. Provide the information necessary to create the new virtual network, and then select **Create**.

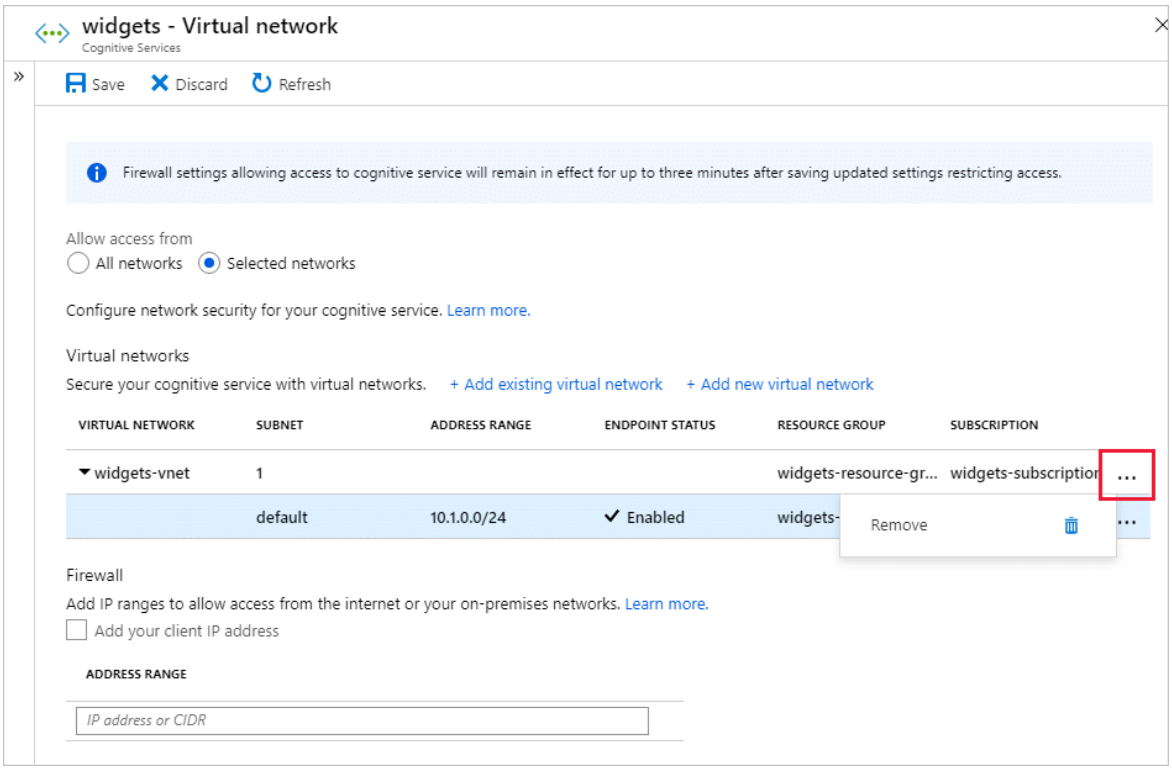


NOTE

If a service endpoint for Azure Cognitive Services wasn't previously configured for the selected virtual network and subnets, you can configure it as part of this operation.

Presently, only virtual networks belonging to the same Azure Active Directory tenant are shown for selection during rule creation. To grant access to a subnet in a virtual network belonging to another tenant, please use Powershell, CLI or REST APIs.

8. To remove a virtual network or subnet rule, select ... to open the context menu for the virtual network or subnet, and select **Remove**.



9. Select **Save** to apply your changes.

> **IMPORTANT**
> Be sure to set the default rule to **deny**, or network rules have no effect.

Grant access from an internet IP range

You can configure Cognitive Services resources to allow access from specific public internet IP address ranges. This configuration grants access to specific services and on-premises networks, effectively blocking general internet traffic.

Provide allowed internet address ranges using CIDR notation in the form `16.17.18.0/24` or as individual IP addresses like `16.17.18.19`.

> **TIP**
> Small address ranges using "/31" or "/32" prefix sizes are not supported. These ranges should be configured using individual IP address rules.

IP network rules are only allowed for **public internet** IP addresses. IP address ranges reserved for private networks (as defined in RFC 1918) aren't allowed in IP rules. Private networks include addresses that start with `10.*`, `172.16.*` - `172.31.*`, and `192.168.*`.

Only IPV4 addresses are supported at this time. Each Cognitive Services resource supports up to 100 IP network rules, which may be combined with Virtual network rules.

Configuring access from on-premises networks

To grant access from your on-premises networks to your Cognitive Services resource with an IP network rule, you must identify the internet facing IP addresses used by your network. Contact your network administrator for help.

If you're using ExpressRoute on-premises for public peering or Microsoft peering, you'll need to identify the NAT

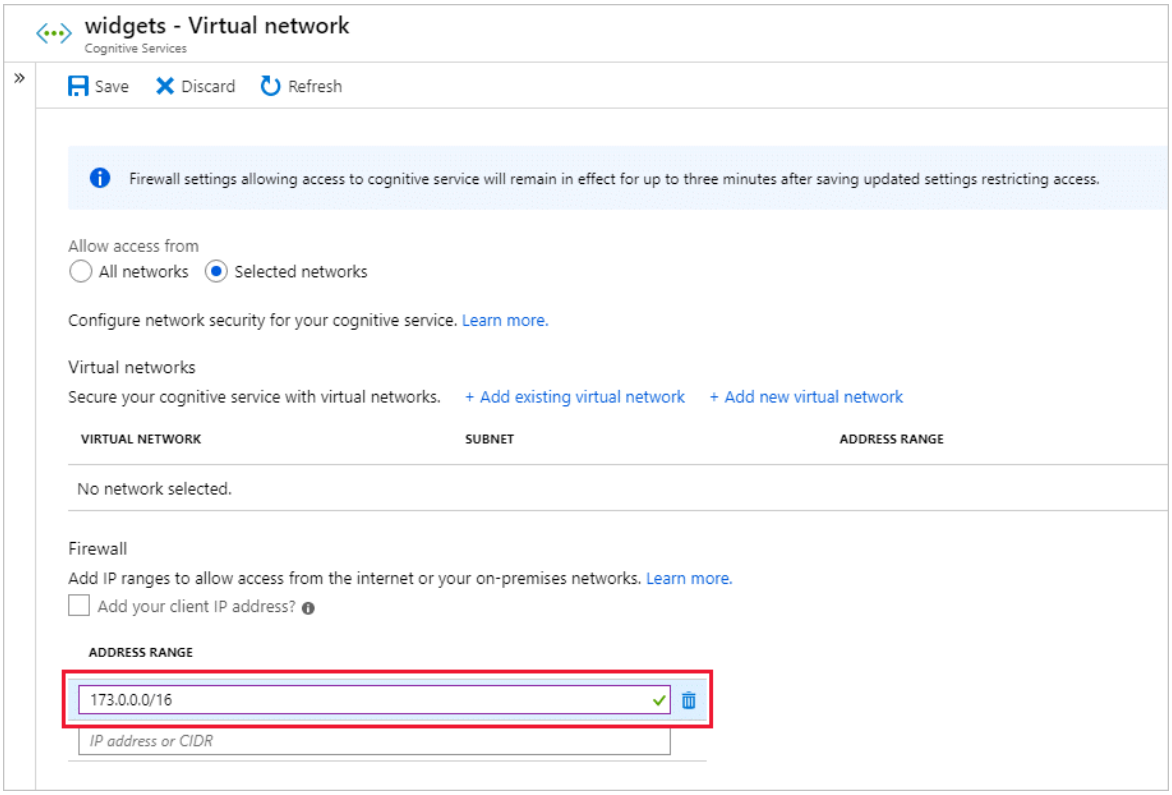
IP addresses. For public peering, each ExpressRoute circuit by default uses two NAT IP addresses. Each is applied to Azure service traffic when the traffic enters the Microsoft Azure network backbone. For Microsoft peering, the NAT IP addresses that are used are either customer provided or are provided by the service provider. To allow access to your service resources, you must allow these public IP addresses in the resource IP firewall setting. To find your public peering ExpressRoute circuit IP addresses, open a support ticket with ExpressRoute via the Azure portal. Learn more about NAT for ExpressRoute public and Microsoft peering.

Managing IP network rules

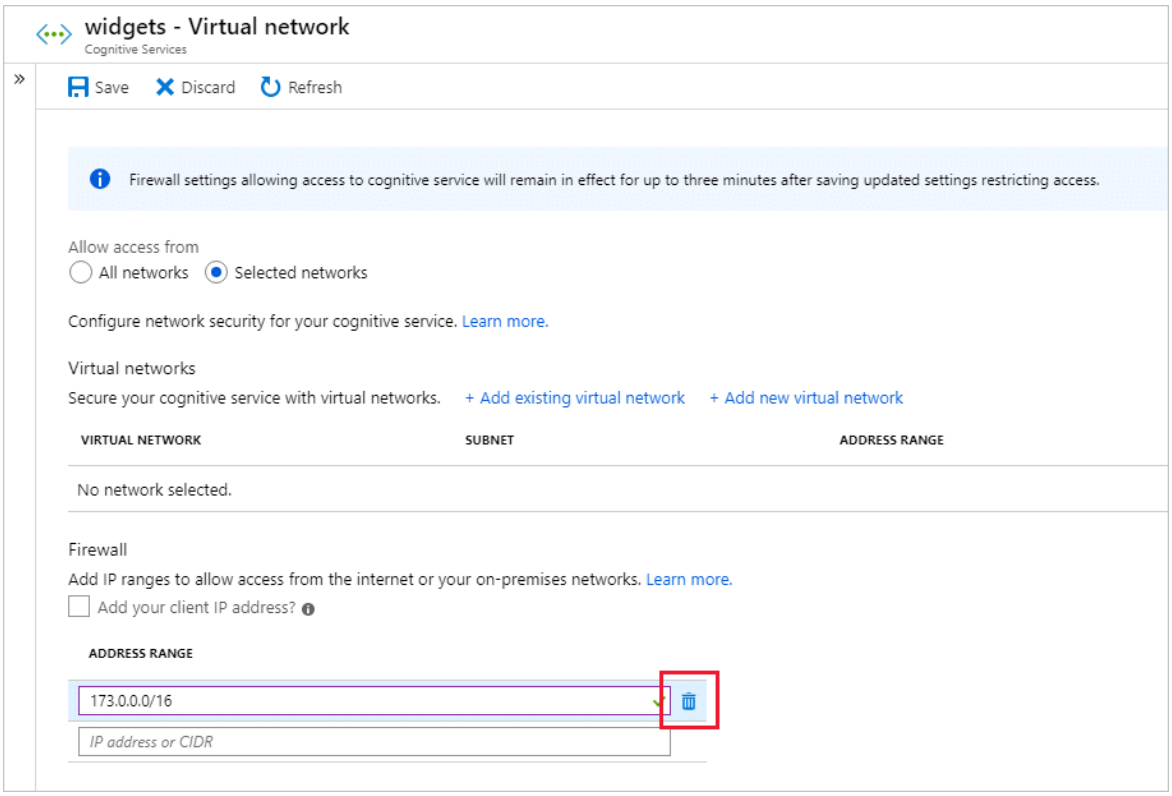
You can manage IP network rules for Cognitive Services resources through the Azure portal, PowerShell, or the Azure CLI.

- Azure portal
- PowerShell
- Azure CLI

1. Go to the Cognitive Services resource you want to secure.
2. Select the **RESOURCE MANAGEMENT** menu called **Virtual network**.
3. Check that you've selected to allow access from **Selected networks**.
4. To grant access to an internet IP range, enter the IP address or address range (in CIDR format) under **Firewall** > **Address Range**. Only valid public IP (non-reserved) addresses are accepted.



5. To remove an IP network rule, select the trash can icon next to the address range.



6. Select **Save** to apply your changes.

> **IMPORTANT**
> Be sure to set the default rule to **deny**, or network rules have no effect.

Use private endpoints

You can use private endpoints for your Cognitive Services resources to allow clients on a virtual network (VNet) to securely access data over a Private Link. The private endpoint uses an IP address from the VNet address space for your Cognitive Services resource. Network traffic between the clients on the VNet and the resource traverses the VNet and a private link on the Microsoft backbone network, eliminating exposure from the public internet.

Private endpoints for Cognitive Services resources let you:

- Secure your Cognitive Services resource by configuring the firewall to block all connections on the public endpoint for the Cognitive Services service.
- Increase security for the VNet, by enabling you to block exfiltration of data from the VNet.
- Securely connect to Cognitive Services resources from on-premises networks that connect to the VNet using VPN or ExpressRoutes with private-peering.

Conceptual overview

A private endpoint is a special network interface for an Azure resource in your VNet. Creating a private endpoint for your Cognitive Services resource provides secure connectivity between clients in your VNet and your resource. The private endpoint is assigned an IP address from the IP address range of your VNet. The connection between the private endpoint and the Cognitive Services service uses a secure private link.

Applications in the VNet can connect to the service over the private endpoint seamlessly, using the same connection strings and authorization mechanisms that they would use otherwise. The exception is the Speech Services, which require a separate endpoint. See the section on Private endpoints with the Speech Services. Private endpoints can be used with all protocols supported by the Cognitive Services resource, including REST.

Private endpoints can be created in subnets that use Service Endpoints. Clients in a subnet can connect to one

Cognitive Services resource using private endpoint, while using service endpoints to access others.

When you create a private endpoint for a Cognitive Services resource in your VNet, a consent request is sent for approval to the Cognitive Services resource owner. If the user requesting the creation of the private endpoint is also an owner of the resource, this consent request is automatically approved.

Cognitive Services resource owners can manage consent requests and the private endpoints, through the '*Private endpoints*' tab for the Cognitive Services resource in the Azure portal.

Private endpoints

When creating the private endpoint, you must specify the Cognitive Services resource it connects to. For more information on creating a private endpoint, see:

- Create a private endpoint using the Private Link Center in the Azure portal
- Create a private endpoint using Azure CLI
- Create a private endpoint using Azure PowerShell

Connecting to private endpoints

Clients on a VNet using the private endpoint should use the same connection string for the Cognitive Services resource as clients connecting to the public endpoint. The exception is the Speech Services, which require a separate endpoint. See the section on Private endpoints with the Speech Services. We rely upon DNS resolution to automatically route the connections from the VNet to the Cognitive Services resource over a private link.

We create a private DNS zone attached to the VNet with the necessary updates for the private endpoints, by default. However, if you're using your own DNS server, you may need to make additional changes to your DNS configuration. The section on DNS changes below describes the updates required for private endpoints.

Private endpoints with the Speech Services

See Using Speech Services with private endpoints provided by Azure Private Link.

DNS changes for private endpoints

When you create a private endpoint, the DNS CNAME resource record for the Cognitive Services resource is updated to an alias in a subdomain with the prefix '*privatelink*'. By default, we also create a private DNS zone, corresponding to the '*privatelink*' subdomain, with the DNS A resource records for the private endpoints.

When you resolve the endpoint URL from outside the VNet with the private endpoint, it resolves to the public endpoint of the Cognitive Services resource. When resolved from the VNet hosting the private endpoint, the endpoint URL resolves to the private endpoint's IP address.

This approach enables access to the Cognitive Services resource using the same connection string for clients in the VNet hosting the private endpoints and clients outside the VNet.

If you are using a custom DNS server on your network, clients must be able to resolve the fully qualified domain name (FQDN) for the Cognitive Services resource endpoint to the private endpoint IP address. Configure your DNS server to delegate your private link subdomain to the private DNS zone for the VNet.

> **TIP**
> When using a custom or on-premises DNS server, you should configure your DNS server to resolve the Cognitive Services resource name in the 'privatelink' subdomain to the private endpoint IP address. You can do this by delegating the 'privatelink' subdomain to the private DNS zone of the VNet, or configuring the DNS zone on your DNS server and adding the DNS A records.

For more information on configuring your own DNS server to support private endpoints, refer to the following articles:

- Name resolution for resources in Azure virtual networks
- DNS configuration for private endpoints

Pricing

For pricing details, see Azure Private Link pricing.

Next steps

- Explore the various Azure Cognitive Services
- Learn more about Azure Virtual Network Service Endpoints

Authenticate requests to Azure Cognitive Services

3/5/2021 • 8 minutes to read • Edit Online

Each request to an Azure Cognitive Service must include an authentication header. This header passes along a subscription key or access token, which is used to validate your subscription for a service or group of services. In this article, you'll learn about three ways to authenticate a request and the requirements for each.

- Authenticate with a single-service or multi-service subscription key
- Authenticate with a token
- Authenticate with Azure Active Directory (AAD)

Prerequisites

Before you make a request, you need an Azure account and an Azure Cognitive Services subscription. If you already have an account, go ahead and skip to the next section. If you don't have an account, we have a guide to get you set up in minutes: Create a Cognitive Services account for Azure.

You can get your subscription key from the Azure portal after creating your account.

Authentication headers

Let's quickly review the authentication headers available for use with Azure Cognitive Services.

HEADER	DESCRIPTION
Ocp-Apim-Subscription-Key	Use this header to authenticate with a subscription key for a specific service or a multi-service subscription key.
Ocp-Apim-Subscription-Region	This header is only required when using a multi-service subscription key with the Translator service. Use this header to specify the subscription region.
Authorization	Use this header if you are using an authentication token. The steps to perform a token exchange are detailed in the following sections. The value provided follows this format: `Bearer <TOKEN>`.

Authenticate with a single-service subscription key

The first option is to authenticate a request with a subscription key for a specific service, like Translator. The keys are available in the Azure portal for each resource that you've created. To use a subscription key to authenticate a request, it must be passed along as the `Ocp-Apim-Subscription-Key` header.

These sample requests demonstrates how to use the `Ocp-Apim-Subscription-Key` header. Keep in mind, when using this sample you'll need to include a valid subscription key.

This is a sample call to the Bing Web Search API:

```
curl -X GET 'https://api.cognitive.microsoft.com/bing/v7.0/search?q=Welsch%20Pembroke%20Corgis' \
-H 'Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY' | json_pp
```

This is a sample call to the Translator service:

```
curl -X POST 'https://api.cognitive.microsofttranslator.com/translate?api-version=3.0&from=en&to=de' \
-H 'Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY' \
-H 'Content-Type: application/json' \
--data-raw '[{ "text": "How much for the cup of coffee?" }]' | json_pp
```

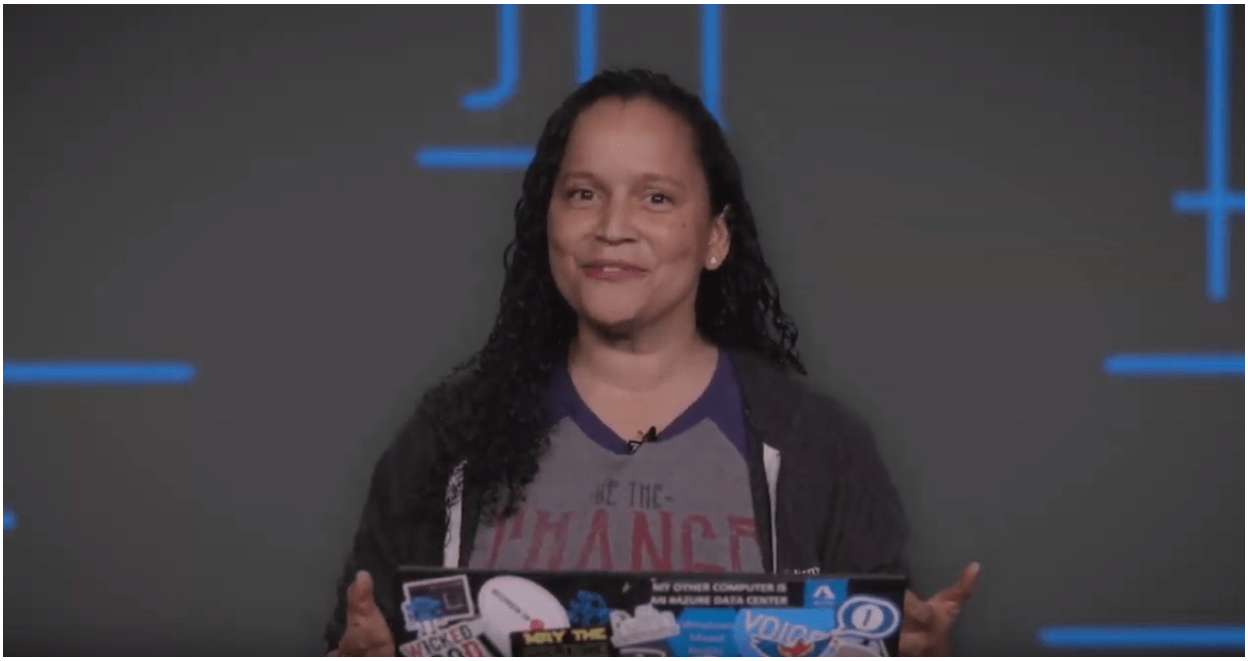
The following video demonstrates using a Cognitive Services key.

Authenticate with a multi-service subscription key

> **WARNING**
>
> At this time, these services **don't** support multi-service keys: QnA Maker, Speech Services, Custom Vision, and Anomaly Detector.

This option also uses a subscription key to authenticate requests. The main difference is that a subscription key is not tied to a specific service, rather, a single key can be used to authenticate requests for multiple Cognitive Services. See Cognitive Services pricing for information about regional availability, supported features, and pricing.

The subscription key is provided in each request as the `Ocp-Apim-Subscription-Key` header.



Supported regions

When using the multi-service subscription key to make a request to `api.cognitive.microsoft.com`, you must include the region in the URL. For example: `westus.api.cognitive.microsoft.com`.

When using multi-service subscription key with the Translator service, you must specify the subscription region with the `Ocp-Apim-Subscription-Region` header.

Multi-service authentication is supported in these regions:

- `australiaeast`
- `brazilsouth`
- `canadacentral`
- `centralindia`

- `eastasia`
- `eastus`
- `japaneast`
- `northeurope`
- `southcentralus`
- `southeastasia`
- `uksouth`
- `westcentralus`
- `westeurope`
- `westus`
- `westus2`

Sample requests

This is a sample call to the Bing Web Search API:

```
curl -X GET 'https://YOUR-REGION.api.cognitive.microsoft.com/bing/v7.0/search?q=Welsch%20Pembroke%20Corgis'
\
-H 'Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY' | json_pp
```

This is a sample call to the Translator service:

```
curl -X POST 'https://api.cognitive.microsofttranslator.com/translate?api-version=3.0&from=en&to=de' \
-H 'Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY' \
-H 'Ocp-Apim-Subscription-Region: YOUR_SUBSCRIPTION_REGION' \
-H 'Content-Type: application/json' \
--data-raw '[{ "text": "How much for the cup of coffee?" }]' | json_pp
```

Authenticate with an authentication token

Some Azure Cognitive Services accept, and in some cases require, an authentication token. Currently, these services support authentication tokens:

- Text Translation API
- Speech Services: Speech-to-text REST API
- Speech Services: Text-to-speech REST API

> **NOTE**
> QnA Maker also uses the Authorization header, but requires an endpoint key. For more information, see QnA Maker: Get answer from knowledge base.

> **WARNING**
> The services that support authentication tokens may change over time, please check the API reference for a service before using this authentication method.

Both single service and multi-service subscription keys can be exchanged for authentication tokens. Authentication tokens are valid for 10 minutes.

Authentication tokens are included in a request as the `Authorization` header. The token value provided must be preceded by `Bearer`, for example: `Bearer YOUR_AUTH_TOKEN`.

Sample requests

Use this URL to exchange a subscription key for an authentication token: `https://YOUR-REGION.api.cognitive.microsoft.com/sts/v1.0/issueToken`.

```
curl -v -X POST \
"https://YOUR-REGION.api.cognitive.microsoft.com/sts/v1.0/issueToken" \
-H "Content-type: application/x-www-form-urlencoded" \
-H "Content-length: 0" \
-H "Ocp-Apim-Subscription-Key: YOUR_SUBSCRIPTION_KEY"
```

These multi-service regions support token exchange:

- `australiaeast`
- `brazilsouth`
- `canadacentral`
- `centralindia`
- `eastasia`
- `eastus`
- `japaneast`
- `northeurope`
- `southcentralus`
- `southeastasia`
- `uksouth`
- `westcentralus`
- `westeurope`
- `westus`
- `westus2`

After you get an authentication token, you'll need to pass it in each request as the `Authorization` header. This is a sample call to the Translator service:

```
curl -X POST 'https://api.cognitive.microsofttranslator.com/translate?api-version=3.0&from=en&to=de' \
-H 'Authorization: Bearer YOUR_AUTH_TOKEN' \
-H 'Content-Type: application/json' \
--data-raw '[{ "text": "How much for the cup of coffee?" }]' | json_pp
```

Authenticate with Azure Active Directory

IMPORTANT

1. Currently, **only** the Computer Vision API, Face API, Text Analytics API, Immersive Reader, Form Recognizer, Anomaly Detector, QnA Maker, and all Bing services except Bing Custom Search support authentication using Azure Active Directory (AAD).
2. AAD authentication needs to be always used together with custom subdomain name of your Azure resource. Regional endpoints does not support AAD authentication.

In the previous sections, we showed you how to authenticate against Azure Cognitive Services using either a single-service or multi-service subscription key. While these keys provide a quick and easy path to start development, they fall short in more complex scenarios that require Azure role-based access control (Azure RBAC). Let's take a look at what's required to authenticate using Azure Active Directory (AAD).

In the following sections, you'll use either the Azure Cloud Shell environment or the Azure CLI to create a subdomain, assign roles, and obtain a bearer token to call the Azure Cognitive Services. If you get stuck, links are provided in each section with all available options for each command in Azure Cloud Shell/Azure CLI.

Create a resource with a custom subdomain

The first step is to create a custom subdomain. If you want to use an existing Cognitive Services resource which does not have custom subdomain name, follow the instructions in Cognitive Services Custom Subdomains to enable custom subdomain for your resource.

1. Start by opening the Azure Cloud Shell. Then select a subscription:

```
Set-AzContext -SubscriptionName <SubscriptionName>
```

2. Next, create a Cognitive Services resource with a custom subdomain. The subdomain name needs to be globally unique and cannot include special characters, such as: ".", "!", ",".

```
$account = New-AzCognitiveServicesAccount -ResourceGroupName <RESOURCE_GROUP_NAME> -name
<ACCOUNT_NAME> -Type <ACCOUNT_TYPE> -SkuName <SUBSCRIPTION_TYPE> -Location <REGION> -
CustomSubdomainName <UNIQUE_SUBDOMAIN>
```

3. If successful, the **Endpoint** should show the subdomain name unique to your resource.

Assign a role to a service principal

Now that you have a custom subdomain associated with your resource, you're going to need to assign a role to a service principal.

> **NOTE**
> Keep in mind that Azure role assignments may take up to five minutes to propagate.

1. First, let's register an AAD application.

```
$SecureStringPassword = ConvertTo-SecureString -String <YOUR_PASSWORD> -AsPlainText -Force

$app = New-AzADApplication -DisplayName <APP_DISPLAY_NAME> -IdentifierUris <APP_URIS> -Password
$SecureStringPassword
```

   You're going to need the **ApplicationId** in the next step.

2. Next, you need to create a service principal for the AAD application.

```
New-AzADServicePrincipal -ApplicationId <APPLICATION_ID>
```

> **NOTE**
> If you register an application in the Azure portal, this step is completed for you.

3. The last step is to assign the "Cognitive Services User" role to the service principal (scoped to the resource). By assigning a role, you're granting service principal access to this resource. You can grant the same service principal access to multiple resources in your subscription.

> **NOTE**
>
> The ObjectId of the service principal is used, not the ObjectId for the application. The ACCOUNT_ID will be the Azure resource Id of the Cognitive Services account you created. You can find Azure resource Id from "properties" of the resource in Azure portal.

```
New-AzRoleAssignment -ObjectId <SERVICE_PRINCIPAL_OBJECTID> -Scope <ACCOUNT_ID> -RoleDefinitionName
"Cognitive Services User"
```

Sample request

In this sample, a password is used to authenticate the service principal. The token provided is then used to call the Computer Vision API.

1. Get your **TenantId**:

```
$context=Get-AzContext
$context.Tenant.Id
```

2. Get a token:

> **NOTE**
>
> If you're using Azure Cloud Shell, the `SecureClientSecret` class isn't available.

- PowerShell
- Azure Cloud Shell

```
$authContext = New-Object "Microsoft.IdentityModel.Clients.ActiveDirectory.AuthenticationContext" -
ArgumentList "https://login.windows.net/<TENANT_ID>"
$secureSecretObject = New-Object "Microsoft.IdentityModel.Clients.ActiveDirectory.SecureClientSecret"
-ArgumentList $SecureStringPassword
$clientCredential = New-Object "Microsoft.IdentityModel.Clients.ActiveDirectory.ClientCredential" -
ArgumentList $app.ApplicationId, $secureSecretObject
$token=$authContext.AcquireTokenAsync("https://cognitiveservices.azure.com/",
$clientCredential).Result
$token
```

3. Call the Computer Vision API:

```
$url = $account.Endpoint+"vision/v1.0/models"
$result = Invoke-RestMethod -Uri $url  -Method Get -Headers
@{"Authorization"=$token.CreateAuthorizationHeader()} -Verbose
$result | ConvertTo-Json
```

Alternatively, the service principal can be authenticated with a certificate. Besides service principal, user principal is also supported by having permissions delegated through another AAD application. In this case, instead of passwords or certificates, users would be prompted for two-factor authentication when acquiring token.

Authorize access to managed identities

Cognitive Services support Azure Active Directory (Azure AD) authentication with managed identities for Azure

resources. Managed identities for Azure resources can authorize access to Cognitive Services resources using Azure AD credentials from applications running in Azure virtual machines (VMs), function apps, virtual machine scale sets, and other services. By using managed identities for Azure resources together with Azure AD authentication, you can avoid storing credentials with your applications that run in the cloud.

Enable managed identities on a VM

Before you can use managed identities for Azure resources to authorize access to Cognitive Services resources from your VM, you must enable managed identities for Azure resources on the VM. To learn how to enable managed identities for Azure Resources, see:

- Azure portal
- Azure PowerShell
- Azure CLI
- Azure Resource Manager template
- Azure Resource Manager client libraries

For more information about managed identities, see Managed identities for Azure resources.

See also

- What is Cognitive Services?
- Cognitive Services pricing
- Custom subdomains

Azure Cognitive Services support and help options

3/20/2021 • 2 minutes to read • Edit Online

Are you just starting to explore the functionality of Azure Cognitive Services? Perhaps you are implementing a new feature in your application. Or after using the service, do you have suggestions on how to improve it? Here are options for where you can get support, stay up-to-date, give feedback, and report bugs for Cognitive Services.

Create an Azure support request

Explore the range of Azure support options and choose the plan that best fits, whether you're a developer just starting your cloud journey or a large organization deploying business-critical, strategic applications. Azure customers can create and manage support requests in the Azure portal.

- Azure portal
- Azure portal for the United States government

Post a question on Microsoft Q&A

For quick and reliable answers on your technical product questions from Microsoft Engineers, Azure Most Valuable Professionals (MVPs), or our expert community, engage with us on Microsoft Q&A, Azure's preferred destination for community support.

If you can't find an answer to your problem using search, submit a new question to Microsoft Q&A. Use one of the following tags when you ask your question:

- Cognitive Services

Vision

- Computer Vision
- Custom Vision
- Face
- Form Recognizer
- Video Indexer

Language

- Immersive Reader
- Language Understanding (LUIS)
- QnA Maker
- Text Analytics
- Translator

Speech

- Speech service

Decision

- Anomaly Detector

- Content Moderator
- Metrics Advisor (preview)
- Personalizer

Post a question to Stack Overflow

For answers on your developer questions from the largest community developer ecosystem, ask your question on Stack Overflow.

If you do submit a new question to Stack Overflow, please use one or more of the following tags when you create the question:

- Cognitive Services

Vision

- Computer Vision
- Custom Vision
- Face
- Form Recognizer
- Video Indexer

Language

- Immersive Reader
- Language Understanding (LUIS)
- QnA Maker
- Text Analytics
- Translator

Speech

- Speech service

Decision

- Anomaly Detector
- Content Moderator
- Metrics Advisor (preview)
- Personalizer

Submit feedback on User Voice

To request new features, post them on UserVoice. Share your ideas for making Cognitive Services and its APIs work better for the applications you develop.

- Cognitive Services

Vision

- Computer Vision
- Custom Vision
- Face

- Form Recognizer
- Video Indexer

Language

- Immersive Reader
- Language Understanding (LUIS)
- QnA Maker
- Text Analytics
- Translator

Speech

- Speech service

Decision

- Anomaly Detector
- Content Moderator
- Metrics Advisor (preview)
- Personalizer

Stay informed

Staying informed about features in a new release or news on the Azure blog can help you find the difference between a programming error, a service bug, or a feature not yet available in Cognitive Services.

- Learn more about product updates, roadmap, and announcements in Azure Updates.
- See what Cognitive Services articles have recently been added or updated in What's new in docs?
- News about Cognitive Services is shared in the Azure blog.
- Join the conversation on Reddit about Cognitive Services.

Next steps

What are Azure Cognitive Services?