

- **AWS IoT credentials (HTTPS, port 443)** – `credentials.iot.<region>.amazonaws.com` and subdomains.
- **Amazon Elastic Container Registry (HTTPS, port 443)** – `api.ecr.<region>.amazonaws.com`, `dkr.ecr.<region>.amazonaws.com` and subdomains.
- **Amazon CloudWatch (HTTPS, port 443)** – `monitoring.<region>.amazonaws.com`.
- **Amazon CloudWatch Logs (HTTPS, port 443)** – `logs.<region>.amazonaws.com`.
- **Amazon Simple Storage Service (HTTPS, port 443)** – `s3.<region>.amazonaws.com`, `s3-accesspoint.<region>.amazonaws.com` and subdomains.

If your application calls other AWS services, the appliance needs access to the endpoints for those services as well. For more information, see [Service endpoints and quotas](#).

Configuring local network access

The appliance needs access to RTSP video streams locally, but not over the internet. Configure your firewall to allow the appliance to access RTSP streams on port 554 internally, and to not allow streams to go out to or come in from the internet.

Local access

- **Real-time streaming protocol (RTSP, port 554)** – To read camera streams.
- **Network time protocol (NTP, port 123)** – To keep the appliance's clock in sync. If you don't run an NTP server on your network, the appliance can also connect to public NTP servers over the internet.

Private connectivity

The AWS Panorama Appliance does not need internet access if you deploy it in a private VPC subnet with a VPN connection to AWS. You can use Site-to-Site VPN or AWS Direct Connect to create a VPN connection between an on-premises router and AWS. Within your private VPC subnet, you create endpoints that let the appliance connect to Amazon Simple Storage Service, AWS IoT, and other services. For more information, see [Connecting an appliance to a private subnet](#).

Managing camera streams in AWS Panorama

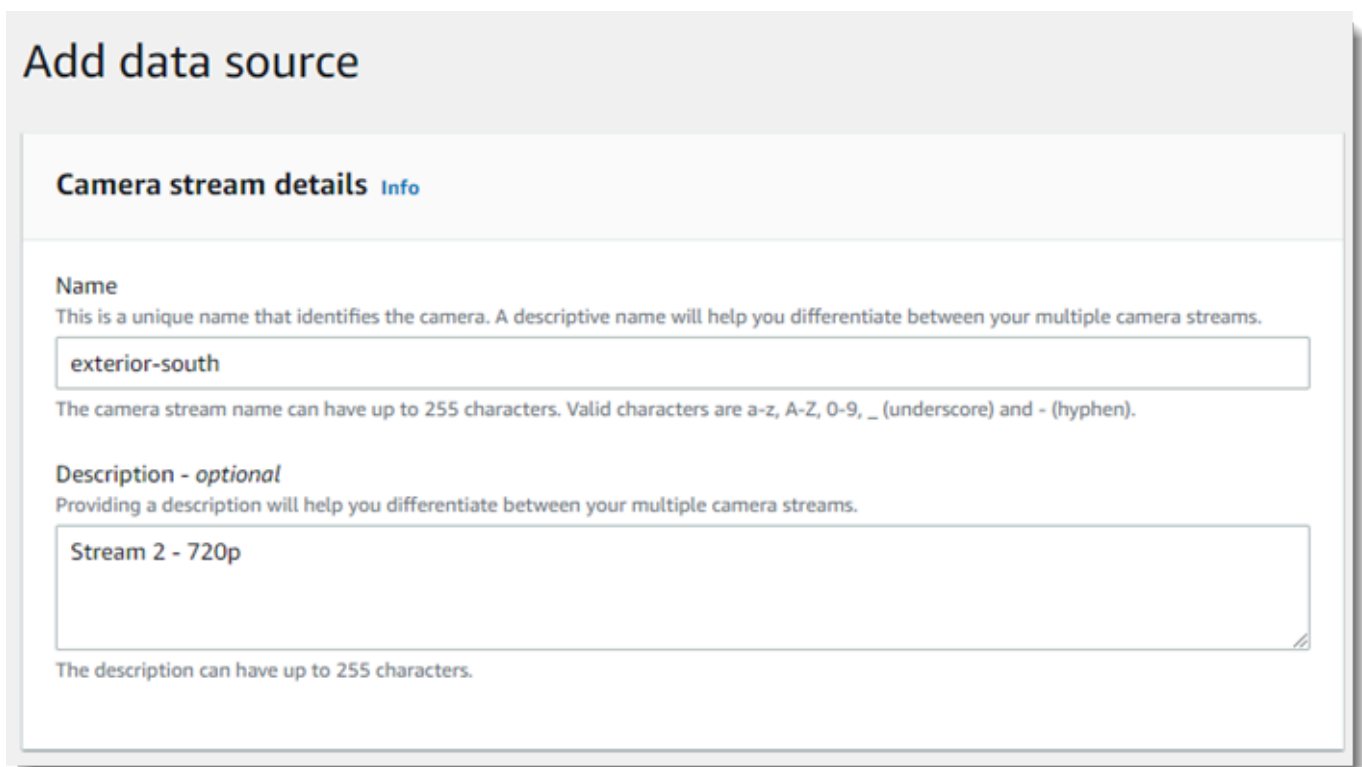
To register video streams as data sources for your application, use the AWS Panorama console. An application can process multiple streams simultaneously and multiple appliances can connect to the same stream.

Important

An application can connect to any camera stream that is routable from the local network it connects to. To secure your video streams, configure your network to allow only RTSP traffic locally. For more information, see [Security in AWS Panorama](#).

To register a camera stream

1. Open the AWS Panorama console [Data sources page](#).
2. Choose **Add data source**.



Add data source

Camera stream details [Info](#)

Name
This is a unique name that identifies the camera. A descriptive name will help you differentiate between your multiple camera streams.

exterior-south

The camera stream name can have up to 255 characters. Valid characters are a-z, A-Z, 0-9, _ (underscore) and - (hyphen).

Description - optional
Providing a description will help you differentiate between your multiple camera streams.

Stream 2 - 720p

The description can have up to 255 characters.

3. Configure the following settings.

- **Name** – A name for the camera stream.

- **Description** – A short description of the camera, its location, or other details.
- **RTSP URL** – A URL that specifies the camera's IP address and the path to the stream. For example, `rtsp://192.168.0.77/live/mpeg4/`
- **Credentials** – If the camera stream is password protected, specify the username and password.

4. Choose **Save**.

To register a camera stream with the AWS Panorama API, see [Automate device registration](#).

For a list of cameras that are compatible with the AWS Panorama Appliance, see [Supported computer vision models and cameras](#).

Removing a stream

You can delete a camera stream in the AWS Panorama console.

To remove a camera stream

1. Open the AWS Panorama console [Data sources page](#).
2. Choose a camera stream.
3. Choose **Delete data source**.

Removing a camera stream from the service does not stop running applications or delete camera credentials from Secrets Manager. To delete secrets, use the [Secrets Manager console](#).

Manage applications on an AWS Panorama Appliance

An application is a combination of code, models, and configuration. From the **Devices** page in the AWS Panorama console, you can manage applications on the appliance.

To manage applications on an AWS Panorama Appliance

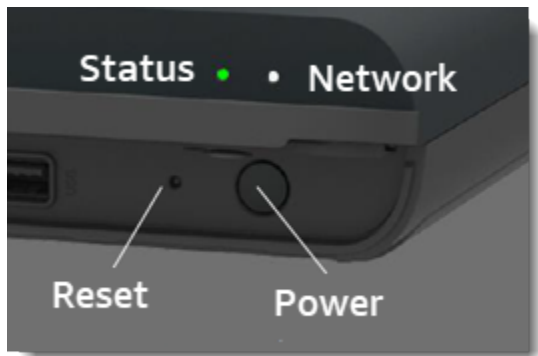
1. Open the AWS Panorama console [Devices page](#).
2. Choose an appliance.

The **Deployed applications** page shows applications that have been deployed to the appliance.

Use the options on this page to remove deployed applications from the appliance, or replace a running application with a new version. You can also clone an application (running or deleted) to deploy a new copy of it.

AWS Panorama Appliance buttons and lights

The AWS Panorama Appliance has two LED lights above the power button that indicate the device status and network connectivity.



Status light

The LEDs change color and blink to indicate status. A slow blink is once every three seconds. A fast blink is once per second.

Status LED states

- **Fast blinking green** – The appliance is booting up.
- **Solid green** – The appliance is operating normally.
- **Slow blinking blue** – The appliance is copying configuration files and attempting to register with AWS IoT.
- **Fast blinking blue** – The appliance is [copying a log image](#) to a USB drive.
- **Fast blinking red** – The appliance encountered an error during startup or is overheated.
- **Slow blinking orange** – The appliance is restoring the latest software version.
- **Fast blinking orange** – The appliance is restoring the minimum software version.

Network light

The network LED has the following states:

Network LED states

- **Solid green** – An Ethernet cable is connected.

- **Blinking green** – The appliance is communicating over the network.
- **Solid red** – An Ethernet cable is not connected.

Power and reset buttons

The power and reset buttons are on the front of the device underneath a protective cover. The reset button is smaller and recessed. Use a small screwdriver or paperclip to press it.

To reset an appliance

1. The appliance must be plugged in and powered off. To power off the appliance, hold the power button for 1 second and wait for the shutdown sequence to complete. The shutdown sequence takes about 10 seconds.
2. To reset the appliance, use the following button combinations. A short press is 1 second. A long press is 5 seconds. For operations that require multiple buttons, press and hold both buttons simultaneously.

- **Full reset** – Long press power and reset.

Restores the minimum software version and deletes all configuration files and applications.

- **Restore latest software version** – Short press reset.

Reapplies the latest software update to the appliance.

- **Restore minimum software version** – Long press reset.

Reapplies the latest required software update to the appliance.

3. Release both buttons. The appliance powers on and the status light blinks orange for several minutes.
4. When the appliance is ready, the status light blinks green.

Resetting an appliance does not delete it from the AWS Panorama service. For more information, see [Deregister an appliance](#).

Managing AWS Panorama applications

Applications run on the AWS Panorama Appliance to perform computer vision tasks on video streams. You can build computer vision applications by combining Python code and machine learning models, and deploy them to the AWS Panorama Appliance over the internet. Applications can send video to a display, or use the AWS SDK to send results to AWS services.

Topics

- [Deploy an application](#)
- [Managing applications in the AWS Panorama console](#)
- [Package configuration](#)
- [The AWS Panorama application manifest](#)
- [Application nodes](#)
- [Application parameters](#)
- [Deploy-time configuration with overrides](#)

Deploy an application

To deploy an application, you use the AWS Panorama Application CLI import it to your account, build the container, upload and register assets, and create an application instance. This topic goes into each of these steps in detail and describes what goes on in the background.

If you have not deployed an application yet, see [Getting started with AWS Panorama](#) for a walkthrough.

For more information on customizing and extending the sample application, see [Building AWS Panorama applications](#).

Sections

- [Install the AWS Panorama Application CLI](#)
- [Import an application](#)
- [Build a container image](#)
- [Import a model](#)
- [Upload application assets](#)
- [Deploy an application with the AWS Panorama console](#)
- [Automate application deployment](#)

Install the AWS Panorama Application CLI

To install the AWS Panorama Application CLI and AWS CLI, use pip.

```
$ pip3 install --upgrade awscli panoramacli
```

To build application images with the AWS Panorama Application CLI, you need Docker. On Linux, qemu and related system libraries are required as well. For more information on installing and configuring the AWS Panorama Application CLI, see the README file in the project's GitHub repository.

- github.com/aws/aws-panorama-cli

For instructions on setting up a build environment in Windows with WSL2, see [Setting up a development environment in Windows](#).

Import an application

If you are working with a sample application or an application provided by a third party, use the AWS Panorama Application CLI to import the application.

```
my-app$ panorama-cli import-application
```

This command renames application packages with your account ID. Package names start with the account ID of the account to which they are deployed. When you deploy an application to multiple accounts, you must import and package the application separately for each account.

For example, this guide's sample application a code package and a model package, each named with a placeholder account ID. The `import-application` command renames these to use the account ID that the CLI infers from your workspace's AWS credentials.

```
/aws-panorama-sample
### assets
### graphs
#   ### my-app
#       ### graph.json
### packages
### 123456789012-SAMPLE\_CODE-1.0
#   ### Dockerfile
#   ### application.py
#   ### descriptor.json
#   ### package.json
#   ### requirements.txt
#   ### squeezenet_classes.json
### 123456789012-SQUEEZENET\_PYTORCH-1.0
### descriptor.json
### package.json
```

123456789012 is replaced with your account ID in the package directory names, and in the application manifest (`graph.json`), which refers to them. You can confirm your account ID by calling `aws sts get-caller-identity` with the AWS CLI.

```
$ aws sts get-caller-identity
{
  "UserId": "AIDAXMPL7W66UC3GFXMPL",
  "Account": "210987654321",
```

```
"Arn": "arn:aws:iam::210987654321:user/devenv"
}
```

Build a container image

Your application code is packaged in a Docker container image, which includes the application code and libraries that you install in your Dockerfile. Use the AWS Panorama Application CLI `build-container` command to build a Docker image and export a filesystem image.

```
my-app$ panorama-cli build-container --container-asset-name code_asset --package-path
packages/210987654321-SAMPLE_CODE-1.0
{
  "name": "code_asset",
  "implementations": [
    {
      "type": "container",
      "assetUri":
"5fa5xmplbc8c16bf8182a5cb97d626767868d3f4d9958a4e49830e1551d227c5.tar.gz",
      "descriptorUri":
"1872xmpl1129481ed053c52e66d6af8b030f9eb69b1168a29012f01c7034d7a8f.json"
    }
  ]
}
Container asset for the package has been succesfully built at
assets/5fa5xmplbc8c16bf8182a5cb97d626767868d3f4d9958a4e49830e1551d227c5.tar.gz
```

This command creates a Docker image named `code_asset` and exports a filesystem to a `.tar.gz` archive in the `assets` folder. The CLI pulls the application base image from Amazon Elastic Container Registry (Amazon ECR), as specified in the application's Dockerfile.

In addition to the container archive, the CLI creates an asset for the package descriptor (`descriptor.json`). Both files are renamed with a unique identifier that reflects a hash of the original file. The AWS Panorama Application CLI also adds a block to the package configuration that records the names of the two assets. These names are used by the appliance during the deployment process.

Example [packages/123456789012-SAMPLE_CODE-1.0/package.json](#) – with asset block

```
{
  "nodePackage": {
    "envelopeVersion": "2021-01-01",
```

```

    "name": "SAMPLE_CODE",
    "version": "1.0",
    "description": "Computer vision application code.",
    "assets": [
      {
        "name": "code_asset",
        "implementations": [
          {
            "type": "container",
            "assetUri":
"5fa5xmplbc8c16bf8182a5cb97d626767868d3f4d9958a4e49830e1551d227c5.tar.gz",
            "descriptorUri":
"1872xmpl1129481ed053c52e66d6af8b030f9eb69b1168a29012f01c7034d7a8f.json"
          }
        ]
      }
    ],
    "interfaces": [
      {
        "name": "interface",
        "category": "business_logic",
        "asset": "code_asset",
        "inputs": [
          {
            "name": "video_in",
            "type": "media"
          }
        ]
      }
    ]
  }
}

```

The name of the code asset, specified in the `build-container` command, must match the value of the `asset` field in the package configuration. In the preceding example, both values are `code_asset`.

Import a model

Your application might have a model archive in its `assets` folder or that you download separately. If you have a new model, an updated model, or updated model descriptor file, use the `add-raw-model` command to import it.

```

my-app$ panorama-cli add-raw-model --model-asset-name model_asset \
--model-local-path my-model.tar.gz \
--descriptor-path packages/210987654321-SQUEEZENET_PYTORCH-1.0/descriptor.json \
--packages-path packages/210987654321-SQUEEZENET_PYTORCH-1.0

```

If you just need to update the descriptor file, you can reuse the existing model in the assets directory. You might need to update the descriptor file to configure features such as floating point precision mode. For example, the following script shows how to do this with the sample app.

Example [util-scripts/update-model-config.sh](#)

```
#!/bin/bash
set -eo pipefail
MODEL_ASSET=fd1axmplacc3350a5c2673adacffab06af54c3f14da6fe4a8be24cac687a386e
MODEL_PACKAGE=SQUEEZENET_PYTORCH
ACCOUNT_ID=$(ls packages | grep -Eo '[0-9]{12}' | head -1)
panorama-cli add-raw-model --model-asset-name model_asset --model-local-path assets/
${MODEL_ASSET}.tar.gz --descriptor-path packages/${ACCOUNT_ID}-${MODEL_PACKAGE}-1.0/
descriptor.json --packages-path packages/${ACCOUNT_ID}-${MODEL_PACKAGE}-1.0
cp packages/${ACCOUNT_ID}-${MODEL_PACKAGE}-1.0/package.json packages/${ACCOUNT_ID}-
${MODEL_PACKAGE}-1.0/package.json.bup
```

Changes to the descriptor file in the model package directory are not applied until you reimport it with the CLI. The CLI updates the model package configuration with the new asset names in-place, similar to how it updates the configuration for the application code package when you rebuild a container.

Upload application assets

To upload and register the application's assets, which include the model archive, container filesystem archive, and their descriptor files, use the `package-application` command.

```
my-app$ panorama-cli package-application
Uploading package SQUEEZENET_PYTORCH
Patch version for the package
5d3cxmplb7113faa1d130f97f619655d8ca12787c751851a0e155e50eb5e3e96
Deregistering previous patch version
e845xmpl8ea0361eb345c313a8dded30294b3a46b486dc8e7c174ee7aab29362
Asset fd1axmplacc3350a5c2673adacffab06af54c3f14da6fe4a8be24cac687a386e.tar.gz already
exists, ignoring upload
upload: assets/87fbxmpl6f18aeae4d1e3ff8bbc6147390feaf47d85b5da34f8374974ecc4aaf.json
to s3://arn:aws:s3:us-east-2:212345678901:accesspoint/
panorama-210987654321-6k75xmpl2jypelgzst7uux62ye/210987654321/nodePackages/
SQUEEZENET_PYTORCH/
binaries/87fbxmpl6f18aeae4d1e3ff8bbc6147390feaf47d85b5da34f8374974ecc4aaf.json
Called register package version for SQUEEZENET_PYTORCH with patch version
5d3cxmplb7113faa1d130f97f619655d8ca12787c751851a0e155e50eb5e3e96
```

...

If there are no changes to an asset file or the package configuration, the CLI skips it.

```
Uploading package SAMPLE_CODE
Patch Version ca91xmplca526fe3f07821fb0c514f70ed0c444f34cb9bd3a20e153730b35d70 already
registered, ignoring upload
Register patch version complete for SQUEEZENET_PYTORCH with patch version
5d3cxmplb7113faa1d130f97f619655d8ca12787c751851a0e155e50eb5e3e96
Register patch version complete for SAMPLE_CODE with patch version
ca91xmplca526fe3f07821fb0c514f70ed0c444f34cb9bd3a20e153730b35d70
All packages uploaded and registered successfully
```

The CLI uploads the assets for each package to an Amazon S3 access point that is specific to your account. AWS Panorama manages the access point for you, and provides information about it through the [DescribePackage](#) API. The CLI uploads the assets for each package to the location provided for that package, and registers them with the AWS Panorama service with the settings described by the package configuration.

Deploy an application with the AWS Panorama console

You can deploy an application with the AWS Panorama console. During the deployment process, you choose which camera streams to pass to the application code, and configure options provided by the application's developer.

To deploy an application

1. Open the AWS Panorama console [Deployed applications page](#).
2. Choose **Deploy application**.
3. Paste the contents of the application manifest, `graph.json`, into the text editor. Choose **Next**.
4. Enter a name and description.
5. Choose **Proceed to deploy**.
6. Choose **Begin deployment**.
7. If your application [uses a role](#), choose it from the drop-down menu. Choose **Next**.
8. Choose **Select device**, and then choose your appliance. Choose **Next**.
9. On the **Select data sources** step, choose **View input(s)**, and add your camera stream as a data source. Choose **Next**.

10. On the **Configure** step, configure any application-specific settings defined by the developer. Choose **Next**.
11. Choose **Deploy**, and then choose **Done**.
12. In the list of deployed applications, choose the application to monitor its status.

The deployment process takes 15-20 minutes. The appliance's output can be blank for an extended period while the application starts. If you encounter an error, see [Troubleshooting](#).

Automate application deployment

You can automate the application deployment process with the [CreateApplicationInstance](#) API. The API takes two configuration files as input. The application manifest specifies the packages used and their relationships. The second file is an overrides file that specifies deploy-time overrides of values in the application manifest. Using an overrides file lets you use the same application manifest to deploy the application with different camera streams, and configure other application-specific settings.

For more information, and example scripts for each of the steps in this topic, see [Automate application deployment](#).

Managing applications in the AWS Panorama console

Use the AWS Panorama console to manage deployed applications.

Sections

- [Update or copy an application](#)
- [Delete versions and applications](#)

Update or copy an application

To update an application, use the **Replace** option. When you replace an application, you can update its code or models.

To update an application

1. Open the AWS Panorama console [Deployed applications page](#).
2. Choose an application.
3. Choose **Replace**.
4. Follow the instructions to create a new version or application.

There is also a **Clone** option that acts similar to **Replace**, but doesn't remove the old version of the application. You can use this option to test changes to an application without stopping the running version, or to redeploy a version that you've already deleted.

Delete versions and applications

To clean up unused application versions, delete them from your appliances.

To delete an application

1. Open the AWS Panorama console [Deployed applications page](#).
2. Choose an application.
3. Choose **Delete from device**.

Package configuration

When you use the AWS Panorama Application CLI command `panorama-cli package-application`, the CLI uploads your application's assets to Amazon S3 and registers them with AWS Panorama. Assets include binary files (container images and models) and descriptor files, which the AWS Panorama Appliance downloads during deployment. To register a package's assets, you provide a separate package configuration file that defines the package, its assets, and its interface.

The following example shows a package configuration for a code node with one input and one output. The video input provides access to image data from a camera stream. The output node sends processed images out to a display.

Example `packages/1234567890-SAMPLE_CODE-1.0/package.json`

```
{
  "nodePackage": {
    "envelopeVersion": "2021-01-01",
    "name": "SAMPLE_CODE",
    "version": "1.0",
    "description": "Computer vision application code.",
    "assets": [
      {
        "name": "code_asset",
        "implementations": [
          {
            "type": "container",
            "assetUri":
"3d9bxmpl1bdb67a3c9730abb19e48d78780b507f3340ec3871201903d8805328a.tar.gz",
            "descriptorUri":
"1872xmpl1129481ed053c52e66d6af8b030f9eb69b1168a29012f01c7034d7a8f.json"
          }
        ]
      }
    ],
    "interfaces": [
      {
        "name": "interface",
        "category": "business_logic",
        "asset": "code_asset",
        "inputs": [
          {
```



```
        "name": "video_in",
        "type": "media"
      }
    ],
    "outputs": [
      {
        "description": "Video stream output",
        "name": "video_out",
        "type": "media"
      }
    ]
  }
}
```

The `assets` section specifies the names of artifacts that the AWS Panorama Application CLI uploaded to Amazon S3. If you import a sample application or an application from another user, this section can be empty or refer to assets that aren't in your account. When you run `panorama-cli package-application`, the AWS Panorama Application CLI populates this section with the correct values.

The AWS Panorama application manifest

When you deploy an application, you provide a configuration file called an application manifest. This file defines the application as a graph with nodes and edges. The application manifest is part of the application's source code and is stored in the graphs directory.

Example graphs/aws-panorama-sample/graph.json

```
{
  "nodeGraph": {
    "envelopeVersion": "2021-01-01",
    "packages": [
      {
        "name": "123456789012::SAMPLE_CODE",
        "version": "1.0"
      },
      {
        "name": "123456789012::SQUEEZENET_PYTORCH_V1",
        "version": "1.0"
      },
      {
        "name": "panorama::abstract_rtsp_media_source",
        "version": "1.0"
      },
      {
        "name": "panorama::hdmi_data_sink",
        "version": "1.0"
      }
    ],
    "nodes": [
      {
        "name": "code_node",
        "interface": "123456789012::SAMPLE_CODE.interface"
      },
      {
        "name": "model_node",
        "interface": "123456789012::SQUEEZENET_PYTORCH_V1.interface"
      },
      {
        "name": "camera_node",
        "interface": "panorama::abstract_rtsp_media_source.rtsp_v1_interface",
        "overridable": true,
        "overrideMandatory": true,

```

```

        "decorator": {
            "title": "IP camera",
            "description": "Choose a camera stream."
        }
    },
    {
        "name": "output_node",
        "interface": "panorama:hdmi_data_sink.hdmi0"
    },
    {
        "name": "log_level",
        "interface": "string",
        "value": "INFO",
        "overridable": true,
        "decorator": {
            "title": "Logging level",
            "description": "DEBUG, INFO, WARNING, ERROR, or CRITICAL."
        }
    }
    ...
],
"edges": [
    {
        "producer": "camera_node.video_out",
        "consumer": "code_node.video_in"
    },
    {
        "producer": "code_node.video_out",
        "consumer": "output_node.video_in"
    },
    {
        "producer": "log_level",
        "consumer": "code_node.log_level"
    }
]
}
}

```

Nodes are connected by edges, which specify mappings between nodes' inputs and outputs. The output of one node connects to the input of another, forming a graph.

JSON schema

The format of application manifest and override documents is defined in a JSON schema. You can use the JSON schema to validate your configuration documents before deploying. The JSON schema is available in this guide's GitHub repository.

- **JSON schema** – [aws-panorama-developer-guide/resources](https://github.com/aws-panorama-developer-guide/resources)

Application nodes

Nodes are models, code, camera streams, output, and parameters. A node has an interface, which defines its inputs and outputs. The interface can be defined in a package in your account, a package provided by AWS Panorama, or a built-in type.

In the following example, `code_node` and `model_node` refer to the sample code and model packages included with the sample application. `camera_node` uses a package provided by AWS Panorama to create a placeholder for a camera stream that you specify during deployment.

Example graph.json – Nodes

```
"nodes": [
  {
    "name": "code_node",
    "interface": "123456789012::SAMPLE_CODE.interface"
  },
  {
    "name": "model_node",
    "interface": "123456789012::SQUEEZENET_PYTORCH_V1.interface"
  },
  {
    "name": "camera_node",
    "interface": "panorama::abstract_rtsp_media_source.rtsp_v1_interface",
    "overridable": true,
    "overrideMandatory": true,
    "decorator": {
      "title": "IP camera",
      "description": "Choose a camera stream."
    }
  }
]
```

Edges

Edges map the output from one node to the input of another. In the following example, the first edge maps the output from a camera stream node to the input of an application code node. The names `video_in` and `video_out` are defined in the node packages' interfaces.

Example graph.json – edges

```
"edges": [
```

```

    {
        "producer": "camera_node.video_out",
        "consumer": "code_node.video_in"
    },
    {
        "producer": "code_node.video_out",
        "consumer": "output_node.video_in"
    },

```

In your application code, you use the `inputs` and `outputs` attributes to get images from the input stream, and send images to the output stream.

Example application.py – Video input and output

```

def process_streams(self):
    """Processes one frame of video from one or more video streams."""
    frame_start = time.time()
    self.frame_num += 1
    logger.debug(self.frame_num)
    # Loop through attached video streams
    streams = self.inputs.video_in.get()
    for stream in streams:
        self.process_media(stream)
    ...
    self.outputs.video_out.put(streams)

```

Abstract nodes

In an application manifest, an abstract node refers to a package defined by AWS Panorama, which you can use as a placeholder in your application manifest. AWS Panorama provides two types of abstract node.

- **Camera stream** – Choose the camera stream that the application uses during deployment.

Package name – panorama::abstract_rtsp_media_source

Interface name – rtsp_v1_interface

- **HDMI output** – Indicates that the application outputs video.

Package name – panorama::hdmi_data_sink

Interface name – hdmi0

The following example shows a basic set of packages, nodes, and edges for an application that processes camera streams and outputs video to a display. The camera node, which uses the interface from the `abstract_rtsp_media_source` package in AWS Panorama, can accept multiple camera streams as input. The output node, which references `hdmi_data_sink`, gives application code access to a video buffer that is output from the appliance's HDMI port.

Example graph.json – Abstract nodes

```
{
  "nodeGraph": {
    "envelopeVersion": "2021-01-01",
    "packages": [
      {
        "name": "123456789012::SAMPLE_CODE",
        "version": "1.0"
      },
      {
        "name": "123456789012::SQUEEZENET_PYTORCH_V1",
        "version": "1.0"
      },
      {
        "name": "panorama::abstract_rtsp_media_source",
        "version": "1.0"
      },
      {
        "name": "panorama::hdmi_data_sink",
        "version": "1.0"
      }
    ],
    "nodes": [
      {
        "name": "camera_node",
        "interface": "panorama::abstract_rtsp_media_source.rtsp_v1_interface",
        "overridable": true,
        "decorator": {
          "title": "IP camera",
          "description": "Choose a camera stream."
        }
      }
    ],
  },
}
```

```
    {
      "name": "output_node",
      "interface": "panorama:hdmi_data_sink.hdmi0"
    }
  ],
  "edges": [
    {
      "producer": "camera_node.video_out",
      "consumer": "code_node.video_in"
    },
    {
      "producer": "code_node.video_out",
      "consumer": "output_node.video_in"
    }
  ]
}
```


Application parameters

Parameters are nodes that have a basic type and can be overridden during deployment. A parameter can have a default value and a *decorator*, which instructs the application's user how to configure it.

Parameter types

- `string` – A string. For example, `DEBUG`.
- `int32` – An integer. For example, `20`
- `float32` – A floating point number. For example, `47.5`
- `boolean` – `true` or `false`.

The following example shows two parameters, a string and a number, which are sent to a code node as inputs.

Example graph.json – Parameters

```
"nodes": [  
  {  
    "name": "detection_threshold",  
    "interface": "float32",  
    "value": 20.0,  
    "overridable": true,  
    "decorator": {  
      "title": "Threshold",  
      "description": "The minimum confidence percentage for a positive  
classification."  
    },  
  },  
  {  
    "name": "log_level",  
    "interface": "string",  
    "value": "INFO",  
    "overridable": true,  
    "decorator": {  
      "title": "Logging level",  
      "description": "DEBUG, INFO, WARNING, ERROR, or CRITICAL."  
    },  
  }  
]
```

```
    }
    ...
  ],
  "edges": [
    {
      "producer": "detection_threshold",
      "consumer": "code_node.threshold"
    },
    {
      "producer": "log_level",
      "consumer": "code_node.log_level"
    }
    ...
  ]
}
```

You can modify parameters directly in the application manifest, or provide new values at deploy-time with overrides. For more information, see [Deploy-time configuration with overrides](#).

Deploy-time configuration with overrides

You configure parameters and abstract nodes during deployment. If you use the AWS Panorama console to deploy, you can specify a value for each parameter and choose a camera stream as input. If you use the AWS Panorama API to deploy applications, you specify these settings with an overrides document.

An overrides document is similar in structure to an application manifest. For parameters with basic types, you define a node. For camera streams, you define a node and a package that maps to a registered camera stream. Then you define an override for each node that specifies the node from the application manifest that it replaces.

Example overrides.json

```
{
  "nodeGraphOverrides": {
    "nodes": [
      {
        "name": "my_camera",
        "interface": "123456789012::exterior-south.exterior-south"
      },
      {
        "name": "my_region",
        "interface": "string",
        "value": "us-east-1"
      }
    ],
    "packages": [
      {
        "name": "123456789012::exterior-south",
        "version": "1.0"
      }
    ],
    "nodeOverrides": [
      {
        "replace": "camera_node",
        "with": [
          {
            "name": "my_camera"
          }
        ]
      }
    ]
  },
}
```

```
    {
      "replace": "region",
      "with": [
        {
          "name": "my_region"
        }
      ]
    },
    "envelopeVersion": "2021-01-01"
  }
}
```

In the preceding example, the document defines overrides for one string parameter and an abstract camera node. The `nodeOverrides` tells AWS Panorama which nodes in this document override which in the application manifest.

Building AWS Panorama applications

Applications run on the AWS Panorama Appliance to perform computer vision tasks on video streams. You can build computer vision applications by combining Python code and machine learning models, and deploy them to the AWS Panorama Appliance over the internet. Applications can send video to a display, or use the AWS SDK to send results to AWS services.

A [model](#) analyzes images to detect people, vehicles, and other objects. Based on images that it has seen during training, the model tells you what it thinks something is, and how confident it is in its guess. You can train models with your own image data or get started with a sample.

The application's [code](#) process still images from a camera stream, sends them to a model, and processes the result. A model might detect multiple objects and return their shapes and location. The code can use this information to add text or graphics to the video, or to send results to an AWS service for storage or further processing.

To get images from a stream, interact with a model, and output video, application code uses [the AWS Panorama Application SDK](#). The application SDK is a Python library that supports models generated with PyTorch, Apache MXNet, and TensorFlow.

Topics

- [Computer vision models](#)
- [Building an application image](#)
- [Calling AWS services from your application code](#)
- [The AWS Panorama Application SDK](#)
- [Running multiple threads](#)
- [Serving inbound traffic](#)
- [Using the GPU](#)
- [Setting up a development environment in Windows](#)

Computer vision models

A *computer vision model* is a software program that is trained to detect objects in images. A model learns to recognize a set of objects by first analyzing images of those objects through training. A computer vision model takes an image as input and outputs information about the objects that it detects, such as the type of object and its location. AWS Panorama supports computer vision models built with PyTorch, Apache MXNet, and TensorFlow.

Note

For a list of pre-built models that have been tested with AWS Panorama, see [Model compatibility](#).

Sections

- [Using models in code](#)
- [Building a custom model](#)
- [Packaging a model](#)
- [Training models](#)

Using models in code

A model returns one or more results, which can include probabilities for detected classes, location information, and other data. The following example shows how to run inference on an image from a video stream and send the model's output to a processing function.

Example [application.py](#) – Inference

```
def process_media(self, stream):
    """Runs inference on a frame of video."""
    image_data = preprocess(stream.image, self.MODEL_DIM)
    logger.debug('Image data: {}'.format(image_data))
    # Run inference
    inference_start = time.time()
    inference_results = self.call({"data":image_data}, self.MODEL_NODE)
    # Log metrics
    inference_time = (time.time() - inference_start) * 1000
```

```

if inference_time > self.inference_time_max:
    self.inference_time_max = inference_time
self.inference_time_ms += inference_time
# Process results (classification)
self.process_results(inference_results, stream)

```

The following example shows a function that processes results from basic classification model. The sample model returns an array of probabilities, which is the first and only value in the results array.

Example [application.py](#) – Processing results

```

def process_results(self, inference_results, stream):
    """Processes output tensors from a computer vision model and annotates a video
    frame."""
    if inference_results is None:
        logger.warning("Inference results are None.")
        return
    max_results = 5
    logger.debug('Inference results: {}'.format(inference_results))
    class_tuple = inference_results[0]
    enum_vals = [(i, val) for i, val in enumerate(class_tuple[0])]
    sorted_vals = sorted(enum_vals, key=lambda tup: tup[1])
    top_k = sorted_vals[::-1][:max_results]
    indexes = [tup[0] for tup in top_k]

    for j in range(max_results):
        label = 'Class [%s], with probability %.3f.' % (self.classes[indexes[j]],
        class_tuple[0][indexes[j]])
        stream.add_label(label, 0.1, 0.1 + 0.1*j)

```

The application code finds the values with the highest probabilities and maps them to labels in a resource file that's loaded during initialization.

Building a custom model

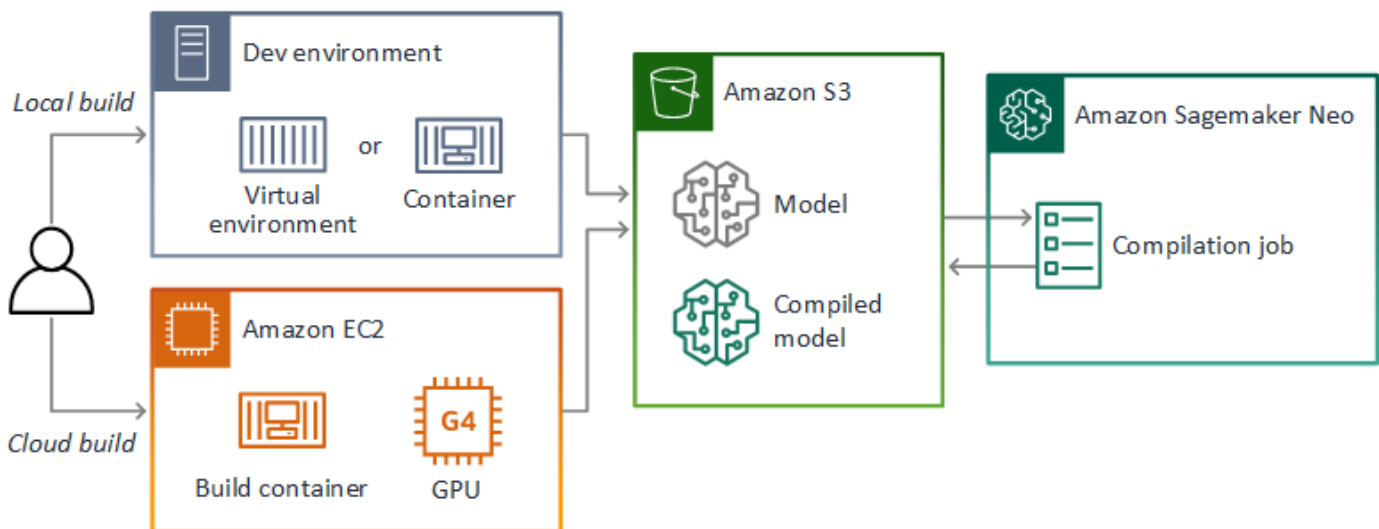
You can use models that you build in PyTorch, Apache MXNet, and TensorFlow in AWS Panorama applications. As an alternative to building and training models in SageMaker AI, you can use a trained model or build and train your own model with a supported framework and export it in a local environment or in Amazon EC2.

Note

For details about the framework versions and file formats supported by SageMaker AI Neo, see [Supported Frameworks](#) in the Amazon SageMaker AI Developer Guide.

The repository for this guide provides a sample application that demonstrates this workflow for a Keras model in TensorFlow SavedModel format. It uses TensorFlow 2 and can run locally in a virtual environment or in a Docker container. The sample app also includes templates and scripts for building the model on an Amazon EC2 instance.

- [Custom model sample application](#)



AWS Panorama uses SageMaker AI Neo to compile models for use on the AWS Panorama Appliance. For each framework, use the [format that's supported by SageMaker AI Neo](#), and package the model in a `.tar.gz` archive.

For more information, see [Compile and deploy models with Neo](#) in the Amazon SageMaker AI Developer Guide.

Packaging a model

A model package comprises a descriptor, package configuration, and model archive. Like in an [application image package](#), the package configuration tells the AWS Panorama service where the model and descriptor are stored in Amazon S3.

Example [packages/123456789012-SQUEEZENET_PYTORCH-1.0/descriptor.json](#)

```
{
  "mlModelDescriptor": {
    "envelopeVersion": "2021-01-01",
    "framework": "PYTORCH",
    "frameworkVersion": "1.8",
    "precisionMode": "FP16",
    "inputs": [
      {
        "name": "data",
        "shape": [
          1,
          3,
          224,
          224
        ]
      }
    ]
  }
}
```

Note

Specify the framework version's major and minor version only. For a list of supported PyTorch, Apache MXNet, and TensorFlow versions versions, see [Supported frameworks](#).

To import a model, use the AWS Panorama Application CLI `import-raw-model` command. If you make any changes to the model or its descriptor, you must rerun this command to update the application's assets. For more information, see [Changing the computer vision model](#).

For the descriptor file's JSON schema, see [assetDescriptor.schema.json](#).

Training models

When you train a model, use images from the target environment, or from a test environment that closely resembles the target environment. Consider the following factors that can affect model performance:

- **Lighting** – The amount of light that is reflected by a subject determines how much detail the model has to analyze. A model trained with images of well-lit subjects might not work well in a low-light or backlit environment.
- **Resolution** – The input size of a model is typically fixed at a resolution between 224 and 512 pixels wide in a square aspect ratio. Before you pass a frame of video to the model, you can downscale or crop it to fit the required size.
- **Image distortion** – A camera's focal length and lens shape can cause images to exhibit distortion away from the center of the frame. The position of a camera also determines which features of a subject are visible. For example, an overhead camera with a wide angle lens will show the top of a subject when it's in the center of the frame, and a skewed view of the subject's side as it moves farther away from center.

To address these issues, you can preprocess images before sending them to the model, and train the model on a wider variety of images that reflect variances in real-world environments. If a model needs to operate in a lighting situations and with a variety of cameras, you need more data for training. In addition to gathering more images, you can get more training data by creating variations of your existing images that are skewed or have different lighting.

Building an application image

The AWS Panorama Appliance runs applications as container filesystems exported from an image that you build. You specify your application's dependencies and resources in a Dockerfile that uses the AWS Panorama application base image as a starting point.

To build an application image, you use Docker and the AWS Panorama Application CLI. The following example from this guide's sample application demonstrates these use cases.

Example [packages/123456789012-SAMPLE_CODE-1.0/Dockerfile](#)

```
FROM public.ecr.aws/panorama/panorama-application
WORKDIR /panorama
COPY . .
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt
```

The following Dockerfile instructions are used.

- **FROM** – Loads the application base image (`public.ecr.aws/panorama/panorama-application`).
- **WORKDIR** – Set the working directory on the image. `/panorama` is used for application code and related files. This setting only persists during the build and does not affect the working directory for your application at runtime (`/`).
- **COPY** – Copies files from a local path to a path on the image. `COPY . .` copies the files in the current directory (the package directory) to the working directory on the image. For example, the application code is copied from `packages/123456789012-SAMPLE_CODE-1.0/application.py` to `/panorama/application.py`.
- **RUN** – Runs shell commands on the image during the build. A single RUN operation can run multiple commands in sequence by using `&&` between commands. This example updates the `pip` package manager and then installs the libraries listed in `requirements.txt`.

You can use other instructions, such as `ADD` and `ARG`, that are useful at build time. Instructions that add runtime information to the container, such as `ENV`, do not work with AWS Panorama. AWS Panorama does not run a container from the image. It only uses the image to export a filesystem, which is transferred to the appliance.

Specifying dependencies

`requirements.txt` is a Python requirements file that specifies libraries used by the application. The sample application uses Open CV and the AWS SDK for Python (Boto3).

Example [packages/123456789012-SAMPLE_CODE-1.0/requirements.txt](#)

```
boto3==1.24.*
opencv-python==4.6.*
```

The `pip install` command in the Dockerfile installs these libraries to the Python `dist-packages` directory under `/usr/local/lib`, so that they can be imported by your application code.

Local storage

AWS Panorama reserves the `/opt/aws/panorama/storage` directory for application storage. Your application can create and modify files at this path. Files created in the storage directory persist across reboots. Other temporary file locations are cleared on boot.

Building image assets

When you build an image for your application package with the AWS Panorama Application CLI, the CLI runs `docker build` in the package directory. This builds an application image that contains your application code. The CLI then creates a container, exports its filesystem, compresses it, and stores it in the assets folder.

```
$ panorama-cli build-container --container-asset-name code_asset --package-path
packages/123456789012-SAMPLE_CODE-1.0
docker build -t code_asset packages/123456789012-SAMPLE_CODE-1.0 --pull
docker export --output=code_asset.tar $(docker create code_asset:latest)
gzip -1 code_asset.tar
{
  "name": "code_asset",
  "implementations": [
    {
      "type": "container",
      "assetUri":
"6f67xmpl132743ed0e60c151a02f2f0da1bf70a4ab9d83fe236fa32a6f9b9f808.tar.gz",
      "descriptorUri":
"1872xmpl1129481ed053c52e66d6af8b030f9eb69b1168a29012f01c7034d7a8f.json"
```

```
    }  
  ]  
}  
Container asset for the package has been succesfully built at /home/  
user/aws-panorama-developer-guide/sample-apps/aws-panorama-sample/  
assets/6f67xmpl32743ed0e60c151a02f2f0da1bf70a4ab9d83fe236fa32a6f9b9f808.tar.gz
```

The JSON block in the output is an asset definition that the CLI adds to the package configuration (package.json) and registers with the AWS Panorama service. The CLI also copies the descriptor file, which specifies the path to the application script (the application's entry point).

Example [packages/123456789012-SAMPLE_CODE-1.0/descriptor.json](#)

```
{  
  "runtimeDescriptor":  
  {  
    "envelopeVersion": "2021-01-01",  
    "entry":  
    {  
      "path": "python3",  
      "name": "/panorama/application.py"  
    }  
  }  
}
```

In the assets folder, the descriptor and application image are named for their SHA-256 checksum. This name is used as a unique identifier for the asset when it is stored in Amazon S3.

Calling AWS services from your application code

You can use the AWS SDK for Python (Boto) to call AWS services from your application code. For example, if your model detects something out of the ordinary, you could post metrics to Amazon CloudWatch, send an notification with Amazon SNS, save an image to Amazon S3, or invoke a Lambda function for further processing. Most AWS services have a public API that you can use with the AWS SDK.

The appliance does not have permission to access any AWS services by default. To grant it permission, [create a role for the application](#), and assign it to the application instance during deployment.

Sections

- [Using Amazon S3](#)
- [Using the AWS IoT MQTT topic](#)

Using Amazon S3

You can use Amazon S3 to store processing results and other application data.

```
import boto3
s3_client=boto3.client("s3")
s3_client.upload_file(data_file,
                      s3_bucket_name,
                      os.path.basename(data_file))
```

Using the AWS IoT MQTT topic

You can use the SDK for Python (Boto3) to send messages to an [MQTT topic](#) in AWS IoT. In the following example, the application posts to a topic named after the appliance's *thing name*, which you can find in [AWS IoT console](#).

```
import boto3
iot_client=boto3.client('iot-data')
topic = "panorama/panorama_my-appliance_Thing_a01e373b"
iot_client.publish(topic=topic, payload="my message")
```

Choose a name that indicates the device ID or other identifier of your choice. To publish messages, the application needs permission to call `iot:Publish`.

To monitor an MQTT queue

1. Open the [AWS IoT console Test page](#).
2. For **Subscription topic**, enter the name of the topic. For example, `panorama/panorama_my-appliance_Thing_a01e373b`.
3. Choose **Subscribe to topic**.

The AWS Panorama Application SDK

The AWS Panorama Application SDK is a Python library for developing AWS Panorama applications. In your [application code](#), you use the AWS Panorama Application SDK to load a computer vision model, run inference, and output video to a monitor.

Note

To ensure that you have access to the latest functionality of the AWS Panorama Application SDK, [upgrade the appliance software](#).

For details about the classes that the application SDK defines and their methods, see [Application SDK reference](#).

Sections

- [Adding text and boxes to output video](#)

Adding text and boxes to output video

With the AWS Panorama SDK, you can output a video stream to a display. The video can include text and boxes that show output from the model, the current state of the application, or other data.

Each object in the `video_in` array is an image from a camera stream that is connected to the appliance. The type of this object is `panoramasdk.media`. It has methods to add text and rectangular boxes to the image, which you can then assign to the `video_out` array.

In the following example, the sample application adds a label for each of the results. Each result is positioned at the same left position, but at different heights.

```
for j in range(max_results):
    label = 'Class [%s], with probability %.3f.' % (self.classes[indexes[j]],
class_tuple[0][indexes[j]])
    stream.add_label(label, 0.1, 0.1 + 0.1*j)
```

To add a box to the output image, use `add_rect`. This method takes 4 values between 0 and 1, indicating the position of the top left and bottom right corners of the box.


```
w,h,c = stream.image.shape  
stream.add_rect(x1/w, y1/h, x2/w, y2/h)
```

Running multiple threads

You can run your application logic on a processing thread and use other threads for other background processes. For example, you can create a thread that [serves HTTP traffic](#) for debugging, or a thread that monitors inference results and sends data to AWS.

To run multiple threads, you use the [threading module](#) from the Python standard library to create a thread for each process. The following example shows the main loop of the debug server sample application, which creates an application object and uses it to run three threads.

Example [packages/123456789012-DEBUG_SERVER-1.0/application.py](#) – Main loop

```
def main():
    panorama = panoramasdk.node()
    while True:
        try:
            # Instantiate application
            logger.info('INITIALIZING APPLICATION')
            app = Application(panorama)
            # Create threads for stream processing, debugger, and client
            app.run_thread = threading.Thread(target=app.run_cv)
            app.server_thread = threading.Thread(target=app.run_debugger)
            app.client_thread = threading.Thread(target=app.run_client)
            # Start threads
            logger.info('RUNNING APPLICATION')
            app.run_thread.start()
            logger.info('RUNNING SERVER')
            app.server_thread.start()
            logger.info('RUNNING CLIENT')
            app.client_thread.start()
            # Wait for threads to exit
            app.run_thread.join()
            app.server_thread.join()
            app.client_thread.join()
            logger.info('RESTARTING APPLICATION')
        except:
            logger.exception('Exception during processing loop.')
```

When all of the threads exit, the application restarts itself. The `run_cv` loop processes images from camera streams. If it receives a signal to stop, it shuts down the debugger process, which runs an HTTP server and can't shut itself down. Each thread must handle its own errors. If an error is not caught and logged, the thread exits silently.

Example [packages/123456789012-DEBUG_SERVER-1.0/application.py](#) – Processing loop

```

# Processing loop
def run_cv(self):
    """Run computer vision workflow in a loop."""
    logger.info("PROCESSING STREAMS")
    while not self.terminate:
        try:
            self.process_streams()
            # turn off debug logging after 15 loops
            if logger.getEffectiveLevel() == logging.DEBUG and self.frame_num ==
15:
                logger.setLevel(logging.INFO)
        except:
            logger.exception('Exception on processing thread.')
    # Stop signal received
    logger.info("SHUTTING DOWN SERVER")
    self.server.shutdown()
    self.server.server_close()
    logger.info("EXITING RUN THREAD")

```

Threads communicate via the application's `self` object. To restart the application processing loop, the debugger thread calls the stop method. This method sets a `terminate` attribute, which signals the other threads to shut down.

Example [packages/123456789012-DEBUG_SERVER-1.0/application.py](#) – Stop method

```

# Interrupt processing loop
def stop(self):
    """Signal application to stop processing."""
    logger.info("STOPPING APPLICATION")
    # Signal processes to stop
    self.terminate = True
# HTTP debug server
def run_debugger(self):
    """Process debug commands from local network."""
    class ServerHandler(SimpleHTTPRequestHandler):
        # Store reference to application
        application = self
        # Get status
        def do_GET(self):
            """Process GET requests."""
            logger.info('Get request to {}'.format(self.path))

```

```
        if self.path == "/status":
            self.send_200('OK')
        else:
            self.send_error(400)
# Restart application
def do_POST(self):
    """Process POST requests."""
    logger.info('Post request to {}'.format(self.path))
    if self.path == '/restart':
        self.send_200('OK')
        ServerHandler.application.stop()
    else:
        self.send_error(400)
```

Serving inbound traffic

You can monitor or debug applications locally by running an HTTP server alongside your application code. To serve external traffic, you map ports on the AWS Panorama Appliance to ports on your application container.

Important

By default, the AWS Panorama Appliance does not accept incoming traffic on any ports. Opening ports on the appliance has implicit security risk. When you use this feature, you must take additional steps to [secure your appliance from external traffic](#) and secure communications between authorized clients and the appliance.

The sample code included with this guide is for demonstration purposes and does not implement authentication, authorization, or encryption.

You can open up ports in the range 8000–9000 on the appliance. These ports, when opened, can receive traffic from any routable client. When you deploy your application, you specify which ports to open, and map ports on the appliance to ports on your application container. The appliance software forwards traffic to the container, and sends responses back to the requestor. Requests are received on the appliance port that you specify and responses go out on a random ephemeral port.

Configuring inbound ports

You specify port mappings in three places in your application configuration. The code package's `package.json`, you specify the port that the code node listens on in a `network` block. The following example declares that the node listens on port 80.

Example [packages/123456789012-DEBUG_SERVER-1.0/package.json](#)

```
"outputs": [  
  {  
    "description": "Video stream output",  
    "name": "video_out",  
    "type": "media"  
  }  
],  
"network": {  
  "inboundPorts": [  
    {
```

```

        "port": 80,
        "description": "http"
      }
    ]
  }
}

```

In the application manifest, you declare a routing rule that maps a port on the appliance to a port on the application's code container. The following example adds a rule that maps port 8080 on the device to port 80 on the code_node container.

Example [graphs/my-app/graph.json](#)

```

{
  "producer": "model_input_width",
  "consumer": "code_node.model_input_width"
},
{
  "producer": "model_input_order",
  "consumer": "code_node.model_input_order"
}
],
"networkRoutingRules": [
  {
    "node": "code_node",
    "containerPort": 80,
    "hostPort": 8080,
    "decorator": {
      "title": "Listener port 8080",
      "description": "Container monitoring and debug."
    }
  }
]
]

```

When you deploy the application, you specify the same rules in the AWS Panorama console, or with an override document passed to the [CreateApplicationInstance](#) API. You must provide this configuration at deploy time to confirm that you want to open ports on the appliance.

Example [graphs/my-app/override.json](#)

```

{
  "replace": "camera_node",
  "with": [

```

```
        {
            "name": "exterior-north"
        }
    ]
},
"networkRoutingRules": [
    {
        "node": "code_node",
        "containerPort": 80,
        "hostPort": 8080
    }
],
"envelopeVersion": "2021-01-01"
}
```

If the device port specified in the application manifest is in use by another application, you can use the override document to choose a different port.

Serving traffic

With ports open on the container, you can open a socket or run a server to handle incoming requests. The debug-server sample shows a basic implementation of an HTTP server running alongside computer vision application code.

Important

The sample implementation is not secure for production use. To avoid making your appliance vulnerable to attacks, you must implement appropriate security controls in your code and network configuration.

Example [packages/123456789012-DEBUG_SERVER-1.0/application.py](#) – HTTP server

```
# HTTP debug server
def run_debugger(self):
    """Process debug commands from local network."""
    class ServerHandler(SimpleHTTPRequestHandler):
        # Store reference to application
        application = self
```

```

# Get status
def do_GET(self):
    """Process GET requests."""
    logger.info('Get request to {}'.format(self.path))
    if self.path == '/status':
        self.send_200('OK')
    else:
        self.send_error(400)
# Restart application
def do_POST(self):
    """Process POST requests."""
    logger.info('Post request to {}'.format(self.path))
    if self.path == '/restart':
        self.send_200('OK')
        ServerHandler.application.stop()
    else:
        self.send_error(400)
# Send response
def send_200(self, msg):
    """Send 200 (success) response with message."""
    self.send_response(200)
    self.send_header('Content-Type', 'text/plain')
    self.end_headers()
    self.wfile.write(msg.encode('utf-8'))
try:
    # Run HTTP server
    self.server = HTTPServer(("", self.CONTAINER_PORT), ServerHandler)
    self.server.serve_forever(1)
    # Server shut down by run_cv loop
    logger.info("EXITING SERVER THREAD")
except:
    logger.exception('Exception on server thread.')

```

The server accepts GET requests at the `/status` path to retrieve some information about the application. It also accepts a POST request to `/restart` to restart the application.

To demonstrate this functionality, the sample application runs an HTTP client on a separate thread. The client calls the `/status` path over the local network shortly after startup, and restarts the application a few minutes later.

Example [packages/123456789012-DEBUG_SERVER-1.0/application.py](#) – HTTP client

```
# HTTP test client
```



```

def run_client(self):
    """Send HTTP requests to device port to demonstrate debug server functions."""
    def client_get():
        """Get container status"""
        r = requests.get('http://{}:{}/status'.format(self.device_ip,
self.DEVICE_PORT))
        logger.info('Response: {}'.format(r.text))
        return
    def client_post():
        """Restart application"""
        r = requests.post('http://{}:{}/restart'.format(self.device_ip,
self.DEVICE_PORT))
        logger.info('Response: {}'.format(r.text))
        return
    # Call debug server
    while not self.terminate:
        try:
            time.sleep(30)
            client_get()
            time.sleep(300)
            client_post()
        except:
            logger.exception('Exception on client thread.')
    # stop signal received
    logger.info("EXITING CLIENT THREAD")

```

The main loop manages the threads and restarts the application when they exit.

Example [packages/123456789012-DEBUG_SERVER-1.0/application.py](#) – Main loop

```

def main():
    panorama = panoramasdk.node()
    while True:
        try:
            # Instantiate application
            logger.info('INITIALIZING APPLICATION')
            app = Application(panorama)
            # Create threads for stream processing, debugger, and client
            app.run_thread = threading.Thread(target=app.run_cv)
            app.server_thread = threading.Thread(target=app.run_debugger)
            app.client_thread = threading.Thread(target=app.run_client)
            # Start threads
            logger.info('RUNNING APPLICATION')
            app.run_thread.start()

```

```
        logger.info('RUNNING SERVER')
        app.server_thread.start()
        logger.info('RUNNING CLIENT')
        app.client_thread.start()
        # Wait for threads to exit
        app.run_thread.join()
        app.server_thread.join()
        app.client_thread.join()
        logger.info('RESTARTING APPLICATION')
    except:
        logger.exception('Exception during processing loop.')
```

To deploy the sample application, see the [instructions in this guide's GitHub repository](#).

Using the GPU

You can access the graphics processor (GPU) on the AWS Panorama Appliance to use GPU-accelerated libraries, or run machine learning models in your application code. To turn on GPU access, you add GPU access as a requirement to the package configuration after building your application code container.

Important

If you enable GPU access, you can't run model nodes in any application on the appliance. For security purposes, GPU access is restricted when the appliance runs a model compiled with SageMaker AI Neo. With GPU access, you must run your models in application code nodes, and all applications on the device share access to the GPU.

To turn on GPU access for your application, update the [package configuration](#) after you build the package with the AWS Panorama Application CLI. The following example shows the `requirements` block that adds GPU access to the application code node.

Example `package.json` with `requirements` block

```
{
  "nodePackage": {
    "envelopeVersion": "2021-01-01",
    "name": "SAMPLE_CODE",
    "version": "1.0",
    "description": "Computer vision application code.",
    "assets": [
      {
        "name": "code_asset",
        "implementations": [
          {
            "type": "container",
            "assetUri":
"eba3xmpl171aa387e8f89be9a8c396416cdb80a717bb32103c957a8bf41440b12.tar.gz",
            "descriptorUri":
"4abdxmpl15a6f047d2b3047adde44704759d13f0126c00ed9b4309726f6bb43400ba9.json",
            "requirements": [
              {
                "type": "hardware_access",
                "inferenceAccelerators": [
```

```
        {
            "deviceType": "nvhost_gpu",
            "sharedResourcePolicy": {
                "policy" : "allow_all"
            }
        }
    ]
}
],
"interfaces": [
    ...
```

Update the package configuration between the build and packaging steps in your development workflow.

To deploy an application with GPU access

1. To build the application container, use the `build-container` command.

```
$ panorama-cli build-container --container-asset-name code_asset --package-path
packages/123456789012-SAMPLE_CODE-1.0
```

2. Add the `requirements` block to the package configuration.
3. To upload the container asset and package configuration, use the `package-application` command.

```
$ panorama-cli package-application
```

4. Deploy the application.

For sample applications that use GPU access, visit the [aws-panorama-samples](#) GitHub repository.

Setting up a development environment in Windows

To build a AWS Panorama application, you use Docker, command-line tools, and Python. In Windows, you can set up a development environment by using Docker Desktop with Windows Subsystem for Linux and Ubuntu. This tutorial walks you through the setup process for a development environment that has been tested with AWS Panorama tools and sample applications.

Sections

- [Prerequisites](#)
- [Install WSL 2 and Ubuntu](#)
- [Install Docker](#)
- [Configure Ubuntu](#)
- [Next steps](#)

Prerequisites

To follow this tutorial, you need a version of Windows that supports Windows Subsystem for Linux 2 (WSL 2).

- Windows 10 version 1903 and higher (Build 18362 and higher) or Windows 11
- Windows features
 - Windows Subsystem for Linux
 - Hyper-V
 - Virtual machine platform

This tutorial was developed with the following software versions.

- Ubuntu 20.04
- Python 3.8.5
- Docker 20.10.8

Install WSL 2 and Ubuntu

If you have Windows 10 version 2004 and higher (Build 19041 and higher), you can install WSL 2 and Ubuntu 20.04 with the following PowerShell command.

```
> wsl --install -d Ubuntu-20.04
```

For older Windows version, follow the instructions in the WSL 2 documentation: [Manual installation steps for older versions](#)

Install Docker

To install Docker Desktop, download and run the installer package from hub.docker.com. If you encounter issues, follow the instructions on the Docker website: [Docker Desktop WSL 2 backend](#).

Run Docker Desktop and follow the first-run tutorial to build an example container.

Note

Docker Desktop only enables Docker in the default distribution. If you have other Linux distributions installed prior to running this tutorial, enable Docker in the newly installed Ubuntu distribution in the Docker Desktop settings menu under **Resources, WSL integration**.

Configure Ubuntu

You can now run Docker commands in your Ubuntu virtual machine. To open a command-line terminal, run the distribution from the start menu. The first time you run it, you configure a username and password that you can use to run administrator commands.

To complete configuration of your development environment, update the virtual machine's software and install tools.

To configure the virtual machine

1. Update the software that comes with Ubuntu.

```
$ sudo apt update && sudo apt upgrade -y && sudo apt autoremove
```

2. Install development tools with apt.

```
$ sudo apt install unzip python3-pip
```

3. Install Python libraries with pip.

```
$ pip3 install awscli panoramacli
```

4. Open a new terminal, and then run `aws configure` to configure the AWS CLI.

```
$ aws configure
```

If you don't have access keys, you can generate them in the [IAM console](#).

Finally, download and import the sample application.

To get the sample application

1. Download and extract the sample application.

```
$ wget https://github.com/awsdocs/aws-panorama-developer-guide/releases/download/v1.0-ga/aws-panorama-sample.zip
$ unzip aws-panorama-sample.zip
$ cd aws-panorama-sample
```

2. Run the included scripts to test compilation, build the application container, and upload packages to AWS Panorama.

```
aws-panorama-sample$ ./0-test-compile.sh
aws-panorama-sample$ ./1-create-role.sh
aws-panorama-sample$ ./2-import-app.sh
aws-panorama-sample$ ./3-build-container.sh
aws-panorama-sample$ ./4-package-app.sh
```

The AWS Panorama Application CLI uploads packages and registers them with the AWS Panorama service. You can now [deploy the sample app](#) with the AWS Panorama console.

Next steps

To explore and edit the project files, you can use File Explorer or an integrated development environment (IDE) that supports WSL.

To access the virtual machine's file system, open File explorer and enter `\\wsl$` in the navigation bar. This directory contains a link to the virtual machine's file system (Ubuntu-20.04) and file systems for Docker's data. Under Ubuntu-20.04, your user directory is at `home\username`.

Note

To access files in your Windows installation from within Ubuntu, navigate to the `/mnt/c` directory. For example, you can list files in your downloads directory by running `ls /mnt/c/Users/windows-username/Downloads`.

With Visual Studio Code, you can edit application code in your development environment and run commands with an integrated terminal. To install Visual Studio Code, visit code.visualstudio.com. After installation, add the [Remote WSL](#) extension.

Windows terminal is an alternative to the standard Ubuntu terminal that you've been running commands in. It supports multiple tabs and can run PowerShell, Command Prompt, and terminals for any other variety of Linux that you install. It supports copy and paste with **Ctrl+C** and **Ctrl+V**, clickable URLs, and other useful improvements. To install Windows Terminal, visit microsoft.com.

The AWS Panorama API

You can use the AWS Panorama service's public API to automate device and application management workflows. With the AWS Command Line Interface or the AWS SDK, you can develop scripts or applications that manage resources and deployments. This guide's GitHub repository includes scripts that you can use as a starting point for your own code.

- [aws-panorama-developer-guide/util-scripts](#)

Sections

- [Automate device registration](#)
- [Manage appliances with the AWS Panorama API](#)
- [Automate application deployment](#)
- [Manage applications with the AWS Panorama API](#)
- [Using VPC endpoints](#)

Automate device registration

To provision an appliance, use the [ProvisionDevice](#) API. The response includes a ZIP file with the device's configuration and temporary credentials. Decode the file and save it in an archive with the prefix `certificates-omni_`.

Example [provision-device.sh](#)

```
if [[ $# -eq 1 ]] ; then
    DEVICE_NAME=$1
else
    echo "Usage: ./provision-device.sh <device-name>"
    exit 1
fi
CERTIFICATE_BUNDLE=certificates-omni_${DEVICE_NAME}.zip
aws panorama provision-device --name ${DEVICE_NAME} --output text --query Certificates
| base64 --decode > ${CERTIFICATE_BUNDLE}
echo "Created certificate bundle ${CERTIFICATE_BUNDLE}"
```

The credentials in the configuration archive expire after 5 minutes. Transfer the archive to your appliance with the included USB drive.

To register a camera, use the [CreateNodeFromTemplateJob](#) API. This API takes a map of template parameters for the camera's username, password, and URL. You can format this map as a JSON document by using Bash string manipulation.

Example [register-camera.sh](#)

```
if [[ $# -eq 3 ]] ; then
    NAME=$1
    USERNAME=$2
    URL=$3
else
    echo "Usage: ./register-camera.sh <stream-name> <username> <rtsp-url>"
    exit 1
fi
echo "Enter camera stream password: "
read PASSWORD
TEMPLATE='{"Username":"MY_USERNAME","Password":"MY_PASSWORD","StreamUrl": "MY_URL"}'
TEMPLATE=${TEMPLATE/MY_USERNAME/$USERNAME}
TEMPLATE=${TEMPLATE/MY_PASSWORD/$PASSWORD}
TEMPLATE=${TEMPLATE/MY_URL/$URL}
```

```
echo ${TEMPLATE}
JOB_ID=$(aws panorama create-node-from-template-job --template-type RTSP_CAMERA_STREAM
--output-package-name ${NAME} --output-package-version "1.0" --node-name ${NAME} --
template-parameters "${TEMPLATE}" --output text)
```

Alternatively, you can load the JSON configuration from a file.

```
--template-parameters file://camera-template.json
```

Manage appliances with the AWS Panorama API

You can automate appliance management tasks with the AWS Panorama API.

View devices

To get a list of appliances with device IDs, use the [ListDevices](#) API.

```
$ aws panorama list-devices
{
  "Devices": [
    {
      "DeviceId": "device-4tafxmplhtmlmzabv5lsacba4ere",
      "Name": "my-appliance",
      "CreatedTime": 1652409973.613,
      "ProvisioningStatus": "SUCCEEDED",
      "LastUpdatedTime": 1652410973.052,
      "LeaseExpirationTime": 1652842940.0
    }
  ]
}
```

To get more details about an appliance, use the [DescribeDevice](#) API.

```
$ aws panorama describe-device --device-id device-4tafxmplhtmlmzabv5lsacba4ere
{
  "DeviceId": "device-4tafxmplhtmlmzabv5lsacba4ere",
  "Name": "my-appliance",
  "Arn": "arn:aws:panorama:us-west-2:123456789012:device/device-4tafxmplhtmlmzabv5lsacba4ere",
  "Type": "PANORAMA_APPLIANCE",
  "DeviceConnectionStatus": "ONLINE",
  "CreatedTime": 1648232043.421,
  "ProvisioningStatus": "SUCCEEDED",
  "LatestSoftware": "4.3.55",
  "CurrentSoftware": "4.3.45",
  "SerialNumber": "GFXMPL0013023708",
  "Tags": {},
  "CurrentNetworkingStatus": {
    "Ethernet0Status": {
      "IpAddress": "192.168.0.1/24",
      "ConnectionStatus": "CONNECTED",
      "HwAddress": "8C:XM:PL:60:C5:88"
    }
  }
}
```

```

    },
    "Ethernet1Status": {
        "IpAddress": "--",
        "ConnectionStatus": "NOT_CONNECTED",
        "HwAddress": "8C:XM:PL:60:C5:89"
    }
},
"LeaseExpirationTime": 1652746098.0
}

```

Upgrade appliance software

If the LatestSoftware version is newer than the CurrentSoftware, you can upgrade the device. Use the [CreateJobForDevices](#) API to create an over-the-air (OTA) update job.

```

$ aws panorama create-job-for-devices --device-ids device-4tafxmplhtmlmzabv5lsacba4ere \
  --device-job-config '{"OTAJobConfig": {"ImageVersion": "4.3.55"}}' --job-type OTA
{
  "Jobs": [
    {
      "JobId": "device-4tafxmplhtmlmzabv5lsacba4ere-0",
      "DeviceId": "device-4tafxmplhtmlmzabv5lsacba4ere"
    }
  ]
}

```

In a script, you can populate the image version field in the job configuration file with Bash string manipulation.

Example [check-updates.sh](#)

```

apply_update() {
  DEVICE_ID=$1
  NEW_VERSION=$2
  CONFIG='{"OTAJobConfig": {"ImageVersion": "NEW_VERSION"}}'
  CONFIG=${CONFIG/NEW_VERSION/$NEW_VERSION}
  aws panorama create-job-for-devices --device-ids ${DEVICE_ID} --device-job-config
  "${CONFIG}" --job-type OTA
}

```

The appliance downloads the specified software version and updates itself. Watch the update's progress with the [DescribeDeviceJob](#) API.