# Analyzing Remote Communications Infrastructure with Computer Vision

To plan wireless networks effectively, Communication Service Providers (CSPs) must analyze the viability of potential cell sites. This includes identifying potential facilities, obstructions, sources of interference, etc. Traditionally, capturing this information required a truck roll, which can be expensive, especially at scale. Harnessing the power of Machine Learning (ML) and computer vision to analyze site images, this AWS solution eliminates the need to visit each location and empowers users by reducing costs and increasing accuracy all while providing additional flexibility as compared to manual labeling.

To keep this post concise, we do not cover all the stages involved in model development and provisioning. Instead, we assume that you already have a model trained. If you would like more information and guidance about how to leverage AWS services to build, train, and deploy your own computer vision endpoints, then you can review [these computer vision resources](https://docs.aws.amazon.com/sagemaker/latest/dg/jumpstart-solutions.html#jumpstart-solutions-computer-vision) in the [Amazon SageMaker](https://aws.amazon.com/sagemaker/) developer guide. You can also view [this Amazon ML post](https://aws.amazon.com/blogs/machine-learning/training-and-deploying-models-using-tensorflow-2-with-the-object-detection-api-on-amazon-sagemaker/), which covers an entire deployment example that includes the use of [Amazon GroundTruth](https://aws.amazon.com/sagemaker/data-labeling/?sagemaker-data-wrangler-whats-new.sort-by=item.additionalFields.postDateTime&sagemaker-data-wrangler-whats-new.sort-order=desc) for labeling training data. You can use the endpoint deployed in that post to test the solution presented here.

## Background

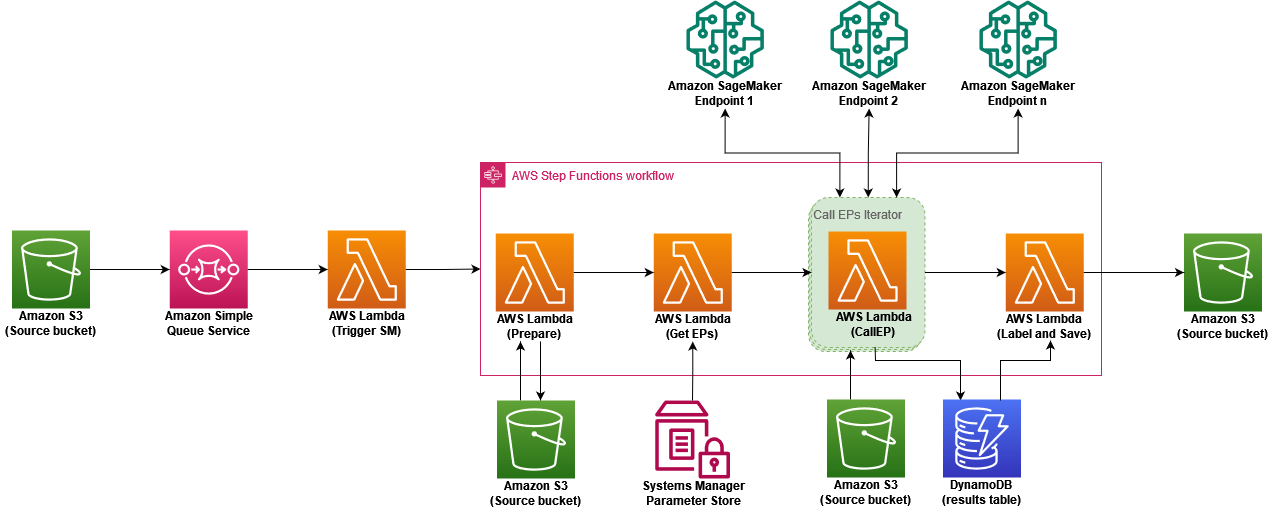
For CSPs, selecting the best location to deploy radio equipment is key to the network’s success. This equipment allows subscribers or consumers to receive the signal that powers their smartphones and data transmissions. However, CSPs often have limited ability to choose and monitor the infrastructure on which to deploy these remote resources. In the United States, tower sites are often rented or leased from tower companies and, occasionally, alternate spaces such as office buildings can also be used. CSPs generally don’t have sufficient information regarding these disparate facilities when deciding where to deploy their infrastructure without first visiting the location or relying on the site inventory and location data provided to them.

To evaluate a location, radio frequency engineering teams need to know factors such as the number and types of antennas already installed on a given tower, surrounding obstructions, and any changes since the last evaluation. In the past, CSPs would need to send a team or manually look through many images of each cell site to evaluate its viability. However, when considering vast geographic areas and a large number of potential locations, collecting this data using manual processes quickly becomes time consuming, costly, and unscalable.

This solution is a starting point that can be expanded to evaluate new locations or monitor existing ones using ML tools to analyze on-the-ground or drone-based photographs of these remote locations. Computer vision rapidly and accurately analyzes tradeoffs between sites with imagery produced by a drone, which is a faster, simpler, and a more cost-effective solution at scale. This process permits data-driven decisions when evaluating competing tower sites and makes inventory information easily accessible across all deployed sites.

## Solution architecture

Images uploaded to an [Amazon Simple Storage Service (Amazon S3)](https://aws.amazon.com/s3/) bucket are automatically analyzed by a customizable number of computer vision models. The results are inserted into an [Amazon DynamoDB](https://aws.amazon.com/dynamodb/) table and used to label the image with bounding boxes, confidence scores, and label name. Finally, a labeled version of the image is uploaded to the S3 bucket.



This solution prefers serverless infrastructure wherever possible to provide computer vision analysis at the lowest cost, while providing high availability and maximum scalability. Currently, the models are deployed on SageMaker instances for optimal performance, but you have the option to utilize [SageMaker Serverless Inference](https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html) to make the solution *entirely serverless*.

## How it works

Images are uploaded to the Amazon S3 source bucket in the folder *raw\_images*. The upload triggers an event that is passed through an [Amazon Simple Queue Service (Amazon SQS)](https://aws.amazon.com/sqs/) queue. When a new message enters the queue, it is passed to the TriggerStateMachine Lambda function, which parses the Amazon SQS message and passes it as an event to a state machine implemented in [AWS Step Functions](https://aws.amazon.com/step-functions/) for processing.

The Step Functions orchestrator has four steps:

1. **Prepare**: The original image is downloaded from Amazon S3, resized, and a copy is saved back to the source bucket with the path *resized\_images*. The code in this function can be modified to include any additional image modifications (color corrections, filters, etc.) We have included [Pillow](https://python-pillow.org/), a popular Python imaging library, as a layer, but this can be replaced or used in conjunction with other image processing libraries.
2. **GetEndpoints**: Retrieves the *endpoint\_config* parameter from the Parameter Store in [AWS Systems Manager](https://aws.amazon.com/systems-manager/). This parameter consists of a string in JSON format containing a list of endpoints with their corresponding parameters as follows:

{

"endpoints\_config": [

{

"ep\_name": "endpoint name",

"db\_key": "key to store object in the database",

"labels": ["labels that the model is looking for"],

"threshold": min confidence score to label image (float)

}

]

}

This parameter allows you to specify an arbitrary number of endpoints with independent configurations. For example, a CSP could deploy two endpoints: one for equipment and another for obstruction detection. In that case, their configuration parameter may look like this:

{

"endpoints\_config": [

{

"ep\_name": "tf2-object-detection-antenna-123456",

"db\_key": "equipment",

"labels": ["Directional", "Radio", "Empty mount"],

"threshold": 0.95

},

{

"ep\_name": "tf2-object-detection-obs-654321",

"db\_key": "obstruction",

"labels": ["Tree", "Building"],

"threshold": 0.88

}

]

}

After retrieving the config parameter, the function parses and passes it as the input to the next state along with the resized image name and Amazon S3 path, bucket name, and the name of the DynamoDB table where the results are saved, which is also retrieved from the Parameter Store.

If your endpoints require additional or different parameters, then you can modify this JSON configuration, the way it is parsed in this state, and the endpoint call in the next state.

1. **CallEndpoints**: This is a [Map](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html) state that receives the array of endpoints and runs each as an input to the CallEndpoint function. For each endpoint, the function downloads the resized image and uses the CV2 library (deployed as an [AWS Lambda](https://aws.amazon.com/lambda/) layer) to convert it into a tensor (i.e., a matrix representation of image information). This tensor is fed to the endpoint for processing through an API call.

The results from the endpoint are then parsed, processed, and stored in the DynamoDB results table. Each table entry contains the image key and one object for each of the endpoints with a list of results. Each item in a list is a dictionary containing the name of the identified item, confidence, and bounding box points as follows:

{

  "image\_name": {

    # A string with the image filename

  },

  "db\_key\_1": { # the value of db\_key in the config parameter

    "L": [

      {

        "M": {

          "boundBoxLTRB": {

# List of box points

          },

          "confScore": {

            # number between 0-1

          },

          "objLabel": {

            # label of the object detected

          }

        }

      },

*// Other detection results of first endpoint...*

    ],

  },

"db\_key\_2": # results for other endpoints

}

For example, after processing an image named *123abc.jpg* through the two endpoints for antennas and obstructions described previously, the results would look like this:

{

  "image\_name": {

    "S": "123abc"

  },

  "equipment": {

    "L": [

      {

        "M": {

          "boundBoxLTRB": {

            "L": [

              {

                "S": "0.586595178"

              },

              # ... other box points

            ]

          },

          "confScore": {

            "S": "0.647898078"

          },

          "objLabel": {

            "S": "Radio"

          }

        }

      },

*// Other detected objects in the equipment endpoint ...*

    ]

  },

  "obstruction": {

    "L": [

      {

        "M": {

          "boundBoxLTRB": {

            # box points as above

          },

          "confScore": {

            "S": "0.89545685"

          },

          "objLabel": {

            "S": "Tree"

          }

        }

      },

*// Other detected objects in the obstruction endpoint ...*

    ]

  }

}

1. **LabelAndSave**: This last step downloads the resized image, retrieves and parses the results from DynamoDB, and uses them to draw boxes and label names. Finally, it stores a labeled version with the path *labeled\_images* in the source S3 bucket.

We are using the Pillow and Font Lambda layers to import these libraries to draw bounding boxes and label names. For example, the color of the bounding box in our code reflects the confidence score for that object. However, you may modify the code or use different libraries if you want different labeling functionality.

## Try it yourself

To use this solution, an AWS account with access and permissions to deploy the following services is necessary:

* [SageMaker](https://aws.amazon.com/sagemaker/)
* [Lambda](https://aws.amazon.com/lambda/)
* [Amazon S3](https://aws.amazon.com/pm/serv-s3/?trk=fecf68c9-3874-4ae2-a7ed-72b6d19c8034&sc_channel=ps&s_kwcid=AL!4422!3!536452728638!e!!g!!amazon%20s3&ef_id=Cj0KCQjwtsCgBhDEARIsAE7RYh25D528BtMo_7MQlSCwtPFC4DtwRLsz9GenfDC5MDUzm-7uTo8WhCoaAvZREALw_wcB:G:s&s_kwcid=AL!4422!3!536452728638!e!!g!!amazon%20s3)
* [Amazon SQS](https://aws.amazon.com/pm/sqs/?ef_id=Cj0KCQjwtsCgBhDEARIsAE7RYh0IbDqCZT4jCFP-Idwaw4ti8WokbNIXkEoD4DN-MjZopamKrGrUpvsaAgXLEALw_wcB:G:s&s_kwcid=AL!4422!3!629393325349!!!g!!)
* [AWS Step Functions](https://aws.amazon.com/pm/step-functions/?ef_id=Cj0KCQjwtsCgBhDEARIsAE7RYh0zwE36wNL1-fH5Xyq654Ry5WgnNLDdBiOENFE6GSstODD14hReP6UaArV8EALw_wcB:G:s&s_kwcid=AL!4422!3!629393325319!!!g!!)
* [DynamoDB](https://aws.amazon.com/dynamodb/?trk=94bf4df1-96e1-4046-a020-b07a2be0d712&sc_channel=ps&s_kwcid=AL!4422!3!610000101513!e!!g!!dynamodb&ef_id=Cj0KCQjwtsCgBhDEARIsAE7RYh36RyGA0XaORfnQZ3YEeWHeX6slwdXwLAPSk1hy3qBVuuKjI_MZwAEaAkMtEALw_wcB:G:s&s_kwcid=AL!4422!3!610000101513!e!!g!!dynamodb)
* [Systems Manager](https://aws.amazon.com/systems-manager/?ef_id=Cj0KCQjwtsCgBhDEARIsAE7RYh3HfUyh6jufx52PHvaV69GHdpMnGky_OQYQ0gebq7IJmIrMYTtR7BUaAkxAEALw_wcB:G:s&s_kwcid=AL!4422!3!629393326000!!!g!!) (Parameter Store)
* [AWS CloudFormation](https://aws.amazon.com/cloudformation/)

### Before you begin

We provide all the necessary code for the functions, and a CloudFormation template that deploys all the required resources for you. This architecture is built around an existing, running endpoint model, so we provide the instructions and resources to launch a sample one before deploying the CloudFormation stack.

If you already have available endpoints, then you may skip ahead to the CloudFormation step. However, since different models differ in their expected inputs and outputs, remember that you probably need to change the code in the functions to fit the input format and parameters of your own endpoints.

### Prepare a bucket with all the provided files

Before deploying the resources, we upload all the source files to an S3 bucket so it is easy to find. You must give it a unique name, but throughout the instructions we refer to this bucket as our *resource bucket*.

You should have the following files:

* *template.yaml*: the CloudFormation template

*deploy\_sample\_endpoint.ipynb*: a Jupyter notebook to deploy an endpoint. You will need a *model.tar.gz*: the parameters for our trained model so we don’t need to train one

* Five Lambda functions
  + *triggerStateMachine.zip*
  + *prepare.zip*
  + *getEndpoints.zip*
  + *callEndpoint.zip*
  + *labelAndSave.zip*
* Three Lambda layers. You will need to [create three layers](https://docs.aws.amazon.com/lambda/latest/dg/creating-deleting-layers.html#layers-create) for the following libraries: [fonts](https://pypi.org/project/fonts/), [pillow](https://pypi.org/project/pillow/), and [opencv-headless](https://pypi.org/project/opencv-python-headless/). Make sure to use the following names:
  + *FontLayer.zip*
  + *OpenCVHeadlessLayer.zip*
  + *PillowLayer.zip*
* *TestImage.zip:* A folder with some images to test the architecture

1. **Create an S3 bucket**
   1. Navigate to Amazon S3, select *buckets,* and select **Create bucket.**
   2. Name it anything you want (must be globally unique). For example, *resources-XXXX-XXXX-XXXX* using your account number.
   3. Leave all the other parameters as default.
   4. Select **Create bucket.**
2. **Upload all the provided documents to your resources bucket**
   1. Once your bucket is deployed, navigate to it and select **Upload.**
   2. Drag all the provided files of select **Add files** and locate them locally.
      * NOTE: The files should NOT be in a folder. They should be uploaded as a “flat” hierarchy.
   3. Leave everything else as default and select **Upload**. The larger files may take a minute or two to upload depending on your internet connection.

### Deploy the object detection sample endpoint with SageMaker

For our sample model, we deploy an already trained model to recognize bees from images.

1. **Provision a SageMaker Notebook Instance**
   1. From your [AWS Management Console](https://aws.amazon.com/console/), navigate to **SageMaker.**
   2. In the left panel go to **Notebook** > **Notebook Instances.**
   3. Select **Create notebook instance** in the top-right corner.
   4. In Notebook instance settings
      * Name your instance (e.g., *sample-endpoint*).
      * Leave the instance type and platform identifiers as default.
   5. In Permissions and encryption
      * Select *Create a new role* in the dropdown and then *Any S3 bucket* in the popup window. Select **Create role.**
      * Leave the other fields as default.
   6. Leave all the optional sections as default and select **Create notebook instance.**
   7. Wait 2-3 minutes for the notebook’s status to go from *Pending* to *InService*, then select **Open JupyterLab.**
2. **Run the provided notebook**
   1. Inside JupyterLab, select the *upload files* icon. It is the upward-pointing arrow at the top of the left panel.
   2. Locate the *deploy\_sample\_endpoint.ipynb* in your local machine and open it. Once it appears in the left panel, double-click the file to open the notebook.
   3. From the dropdown, select **conda\_tensorflow2\_p310** for the Kernel and select **Select.**
   4. Note that you need to copy the Amazon S3 URI of the *model.tar.gz* that you uploaded earlier and paste it into the model\_artifact line in the second cell.
   5. Run the first four cells of notebook but do not run the last cell so you DON’T DELETE THE ENDPOINT.
      * Note that, depending on your [AWS Region](https://aws.amazon.com/about-aws/global-infrastructure/regions_az/), the ml.m5.xlarge instance may not be available. If that's the case and you get an error, then try a similar instance. For example, ml.g5.2xlarge.
   6. Back in the Console, navigate back to SageMaker and in the left panel select **Inference > Endpoints** to see the endpoint being deployed. Wait until the endpoint status goes from *Creating* to *InService*.
   7. Copy the *Name* (not the ARN) of the endpoint and save it somewhere or continue the next steps on a separate window so you have the name available.

### Launch the CloudFormation template to deploy the architecture

1. In the Console, navigate to CloudFormation.
2. Select **Create stack.**
3. Select **Template is ready** and for the source choose **Amazon S3 URL.**
4. Locate the *template.yaml* file in the resources bucket and copy its URL (not the URI), then paste it into the Amazon S3 URL field.
5. Select **Next.**
6. In *Specify stack details*
   1. Provide a name for your stack, such as ComputerVisionStack.
   2. **EndpointConfigValue**: Paste your endpoint name inside the appropriate place in the EndpointConfigValue parameter. It looks like this:

{… "ep\_name": "tf2-object-detection-xxxx-xx-xx-xx-xx-xx-xxx"…}

* 1. **ResourcesPublicBucketName**: This is the name of your resource bucket. Update it.
  2. **SourceBucketName**: This is the name of the source bucket that CloudFormation creates, so it must be a unique name. We upload the images that we want labeled to this bucket.
  3. **TableName**: The name for the DynamoDB table for the model results. You can leave the default name or provide another one.

1. Select **Next.**
2. For stack options, leave everything default and select **Next.**
3. In Review, acknowledge and select **Submit.**
4. Wait until the stack creates.

### Test the solution

1. Navigate to Amazon S3.
2. Find and navigate to the newly created source bucket. It is the source bucket you named in the previous step.
3. Select **Create folder** and call it *raw\_images*.
4. Select **Create folder.**
5. In the *raw\_images* folder, upload one of the test images (or multiple if you prefer). After a few seconds, you should see two folders created in the bucket: *resized\_images/* and *labeled\_images/* with the corresponding results inside.
6. You can examine the endpoint output in the results table by navigating to *DynamoDB > Tables > Explore items* and selecting the results table.
7. You can examine the state of the state machine by navigating to *Step Functions > State machines > ComputerVisionOrchestrator*.

Don’t forget to [delete the stack](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-delete-stack.html) and SageMaker resources after you are done so that you don’t incur unwanted charges. For SageMaker you just need to finish running the last cell to delete the endpoint and then delete the notebook instance.

Congratulations on deploying the solution!



**JP Fagan Manager, Drone/3D Data Team | DISH Wireless**

*“Using computer vision to analyze our network infrastructure instead of manual processes, we project a reduction in site survey costs by 30 percent. The accuracy provided by the AWS solution also gives us confidence that we’re making data-driven decisions when deploying* *and modifying equipment. Eliminating truck rolls and manual image review, we’ve been able to redirect resources to more strategic initiatives that improve our competitive positioning. AWS has this solution increasing our agility and accelerating our time-to-market.”*