Voice AI Connector Guide for Implementors

Version E1A

Released by

Verizon – ACPD

Contents

Revision History

Date	Version	Revision Description	SME	Author
2025 Jun 13	E1A	Initial Document Draft	Brian Badger	Brian Badger

Introduction:

This document provides background and implementation instructions for our voice bot connectors for both Google Customer Engagement Suite with Google AI and for Azure Voicebots with corresponding speech API's. Our fully redundant Interactive Voice Response (IVR) platform, with media relay capabilities, can take direction on customer intent and treatment from your AI applications with our AI integration connector.

To purchase a Voice AI Connector, work with your Verizon Sales Executive to purchase IP Contact Center (IPCC). Specifically, the IPIVR Premium option must be ordered. When purchasing this option, your accompanying SOW will provide you with Verizon Consulting resources to:

- Build your call plan as defined based on your specific requirements
- Help to manage the project, coordinating Verizon work with your own work with your AI provider and the connector.
- Provide a source of knowledge about the connector to help with any configuration and initial troubleshooting questions
- Development and implementation of your test plan when all the development work is deemed complete.
- Gain your sign-off of project completion and move to production.

Once you are in production, any concerns or issues that may arise should first be addressed by your staff to ensure it is not an AI or connector configuration issue. Once this is assessed and you need Verizon support, please follow your standard IP Contact Center support process by placing a ticket with the Verizon Enterprise Center. Your IPCC Welcome Kit, found here, can provide you with further information on placing a support ticket.

1. Voice AI Connector example using Google Dialogflow CX, Google Text-To-Speech and Google Speech-To-Text

In this example, Google App Engine will host the Voice AI Connector for Google Cloud Platform written in Node 22.. It will provide communications between the Voice AI Connector network function and Google Dialogflow CX, Google Text-To-Speech and Google Speech-To-Text.

https://www.verizon.com/business/supportfiles/voice_ai_connector_examples_2025-09-02.tar

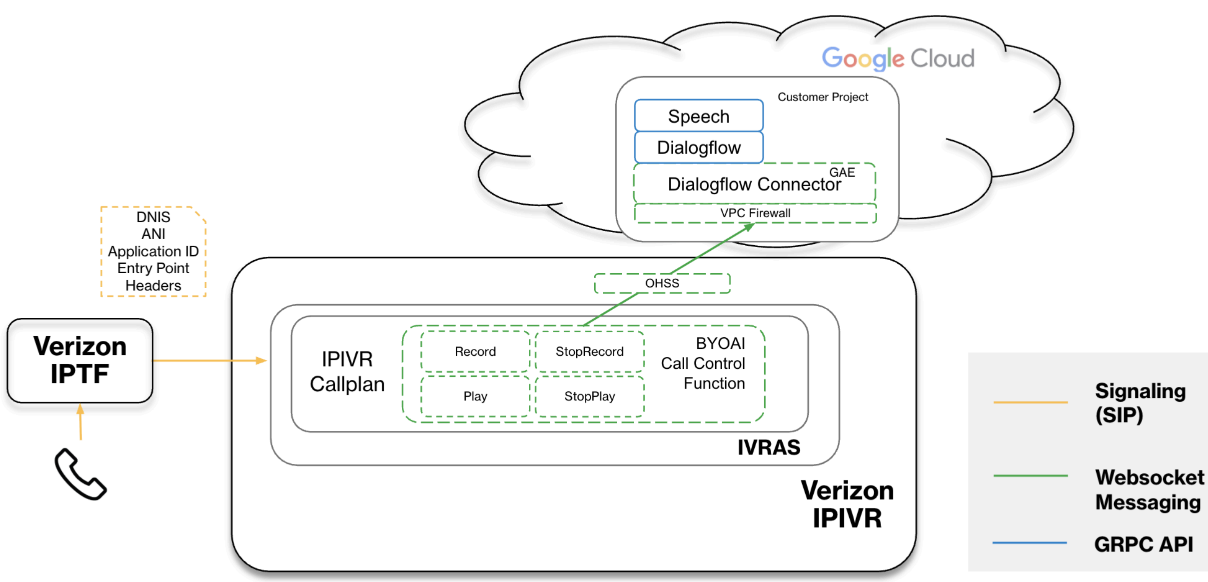
This provides automatic management of Google speech API access control without the use of credentials.

Other hosting solutions are possible, but hosting outside of the Google Cloud Platform ecosystem may require separate management of API credentials.

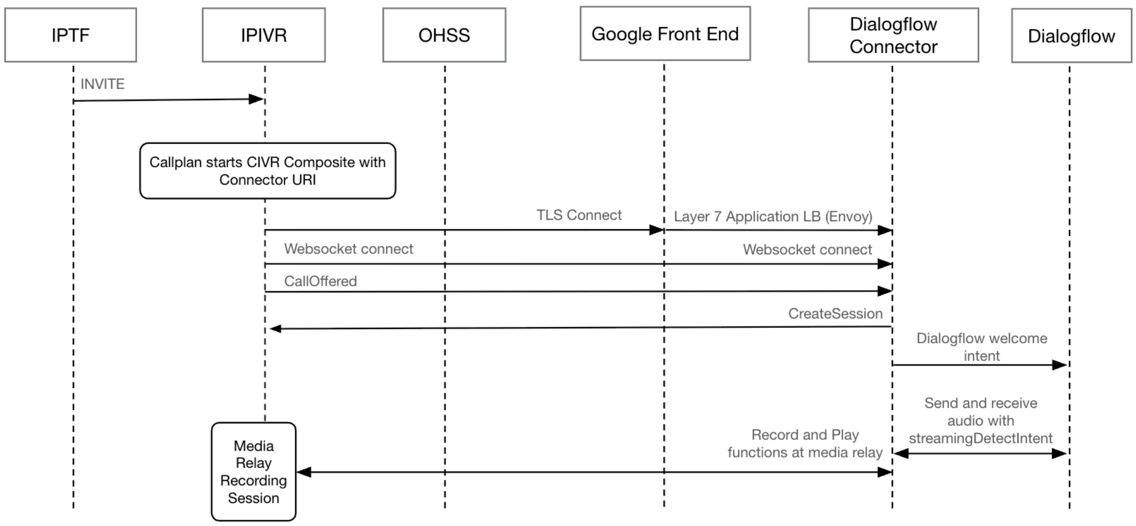
In addition to the steps outlined below, the Google Cloud Platform project where App Engine is deployed must also enable Dialogflow CX with speech-to-text and text-to-speech, and at least one Dialogflow agent must have been built and deployed. One connector instance can support multiple Dialogflow agents and multiple languages within the project.

Geographic redundancy can be achieved by duplicating this design in multiple Google Cloud Platform projects, each within a different geographical region. If multiple connectors are deployed in this way, work with your Verizon representative to ensure that your Verizon IPIVR callplans are updated to implement failover and load-balancing logic between these regions.

1.1. Solution Summary



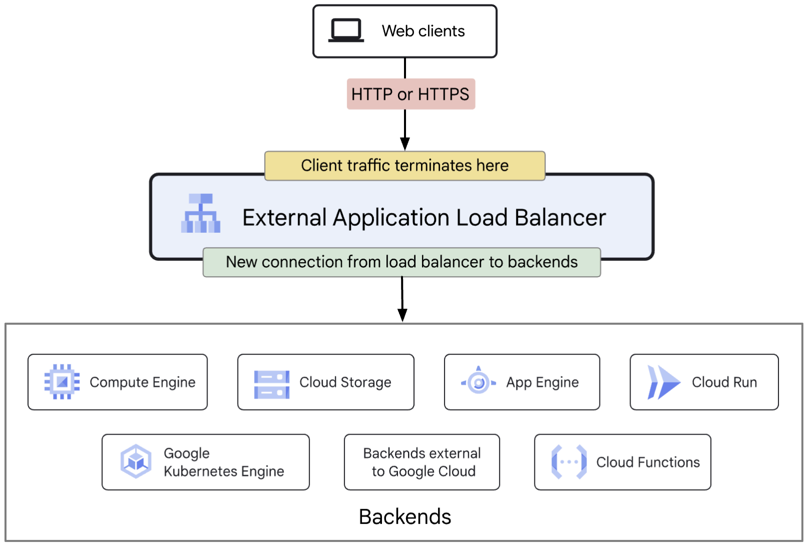
On call arrival, the Voice AI network function connects to an external DialogFlow Connector running in Google App Engine. This Voice AI Connector connects to Dialogflow and directs the call behavior through websocket messages sent to the Voice AI network function, which are then relayed between the media relay and the Voice AI Connector. Media is recorded by the media relay and sent in websocket messages to the connector, which uses Google Dialogflow speech-enabled APIs for transcription and natural language understanding.



The application logic exists in the Dialogflow Connector and the Dialogflow instance, while the Voice AI network function acts as a relay of the messages.

Inbound Voice AI websocket traffic at Google App Engine is terminated at a Google Front End (GFE) as a TLS server, where the Google External Application Load Balancer then routes to the Google App Engine container running on a Google Compute Engine instance using an Envoy HTTP/2 API stream to the App Engine container envoy proxy sidecar.

Websocket traffic is encrypted end-to-end from the Voice AI network function to the Google External Application Load Balancer as HTTPS (external to Google), and encrypted end-to-end from the Google External Application Load Balancer to the App Engine container using HTTP/2 (internal to Google).



External Application Load Balancer architecture.

1.1.1. Deployment procedures

To deploy the connector to Google App Engine to a new project, follow these steps using git and the Google Cloud CLI tool:

Run npm update

```
None
npm update
```

Configure the Google App Engine app.yaml file for the project, VPC (if any), instance class and runtime environment:

```
None
runtime: nodejs
env: flex
runtime_config:
  operating_system: "ubuntu22"
  runtime_version: "20"
service_account: <SERVICE ACCOUNT>@<PROJECT ID>.iam.gserviceaccount.com
service: google-dialogflow-cx
inbound_services:
- warmup
instance_class: f1
handlers:
- url: /.*
```

```
secure: always
redirect_http_response_code: 301
script: auto
```

Configure the service for the Google project, location, agent and language by editing config.json file

```None
...
"projectId": "PROJECT_ID",
"location": "global",
"agentId": "AGENT_ID",
"languageCode": "en-US",
 "ssmlGender":"SSML_VOICE_GENDER_FEMALE",
...
```

Create the Google App Engine instance for the project, region and service account

Before deployment, ensure that the Google project service account has the following IAM roles:

```None
App Engine Deployer
App Engine flexible environment Service Agent
Artifact Registry Reader
Dialogflow API Client
Logs Writer
Monitoring Metric Writer
Storage Object Viewer
```

Then create the App Engine service, firewall settings, and deploy:

```None
gcloud auth login
gcloud config set project $PROJECT
```

```
gcloud app create --region=$REGION
--service-account=$SERVICEACCOUNT@$PROJECT.iam.gserviceaccount.co
m
gcloud app firewall-rules create 1 --action=ALLOW
--source-range=166.35.66.78
gcloud app firewall-rules create 2 --action=ALLOW
--source-range=166.50.89.77
gcloud app firewall-rules create 3 --action=ALLOW
--source-range=166.46.163.188
gcloud app firewall-rules create 4 --action=ALLOW
--source-range=166.50.88.5
gcloud app firewall-rules create 5 --action=ALLOW
--source-range=166.50.29.189
gcloud app firewall-rules create 6 --action=ALLOW
--source-range=166.35.65.2
gcloud app firewall-rules create 7 --action=ALLOW
--source-range=166.40.100.100
gcloud app firewall-rules create 8 --action=ALLOW
--source-range=165.122.81.100
gcloud app firewall-rules create 9 --action=ALLOW
--source-range=152.189.67.133
gcloud app firewall-rules create 10 --action=ALLOW
--source-range=152.189.67.181
gcloud app firewall-rules create 11 --action=ALLOW
--source-range=152.189.74.5
gcloud app firewall-rules create 12 --action=ALLOW
--source-range=152.189.74.53
gcloud app firewall-rules update 2147483647 --action=DENY
gcloud app deploy
```

To deploy the connector in a geo-redundant model, repeat these procedures for each region.

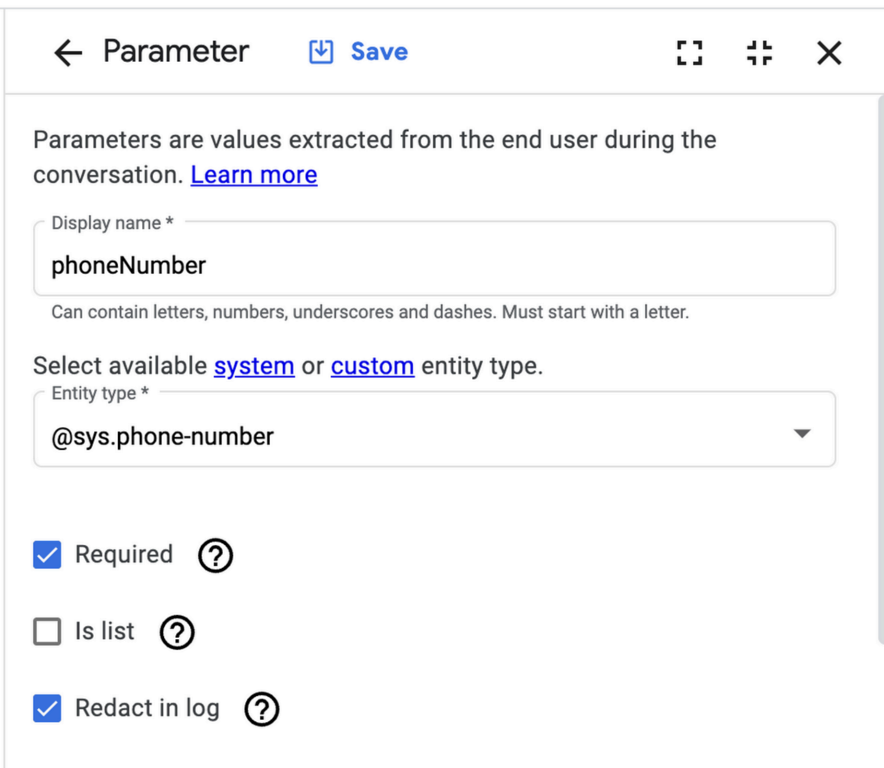
1.1.2. Security Considerations for Google CCAI

While specific security requirements will be driven by the customer application (for example, a customer application that handles PCI data will have PCI requirements, etc), general data handling procedures should be observed by the application developer at all stages.

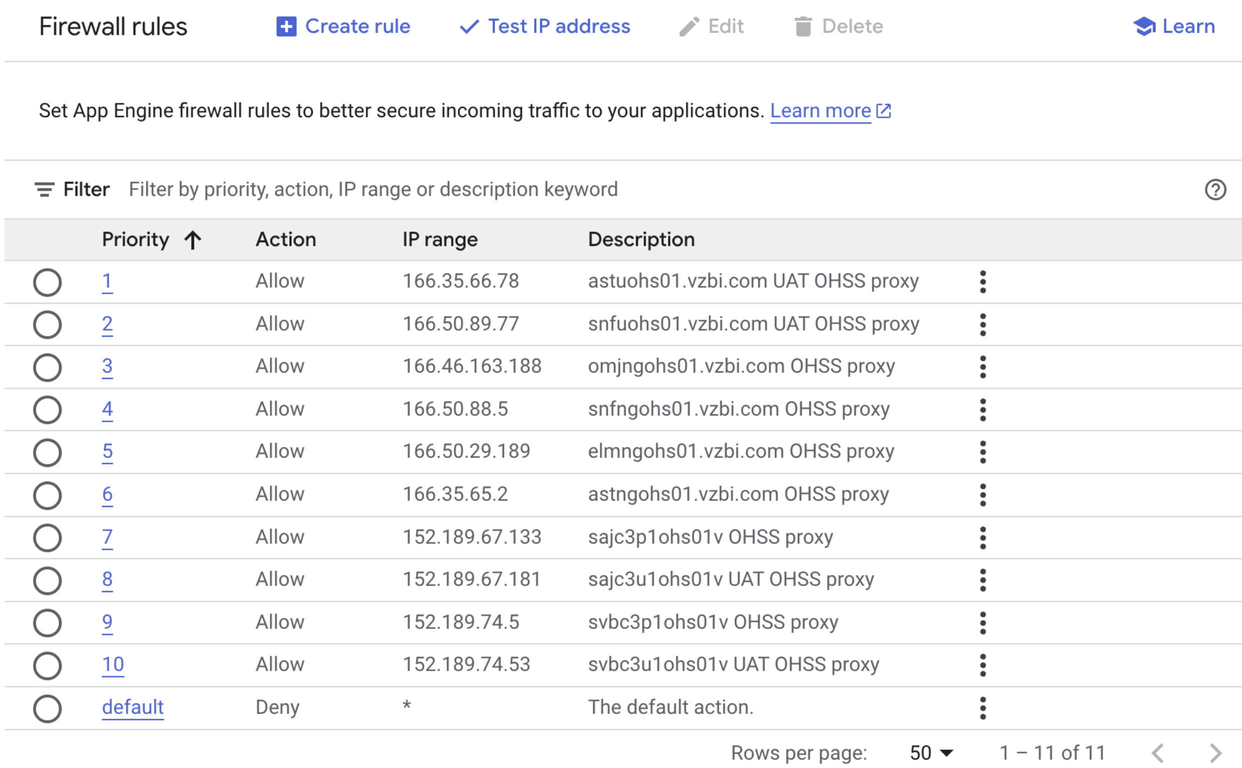
First, data that is carried over the connector APIs – in both directions – must be treated as sensitive by default. **Logging at the callplan and connector must be disabled by default, and data sent and received over the connector APIs should be treated as sensitive by default.**

The data carried over this path is comprised of 1) call arrival metadata including caller identification (ANI), dialed number information (DNIS), and call verification data (STIR/SHAKEN), 2) recorded caller utterances, 3) synthesized application prompts, and 4) Dialogflow Intent Engine final results including application-specified parameters.

Handling of sensitive data within the Dialogflow agent itself can be implemented by marking all sensitive fields as confidential to enable Data Loss Prevention logic which will redact the sensitive data in all logging within Google APIs. Any DLP requirement must be satisfied by the application developer within the Dialogflow project as it cannot be accomplished externally by the connector.



Access to the Dialogflow Connector is restricted to the Verizon OHSS IP addresses listed in Section 4 using the App Engine firewall. This prevents access to the connector from IP addresses not listed.



Firewall rules | Create rule | Test IP address | Edit | Delete | Learn

Set App Engine firewall rules to better secure incoming traffic to your applications. Learn more

Filter | Filter by priority, action, IP range or description keyword

Priority	Action	IP range	Description
1	Allow	166.35.66.78	astuohs01.vzbi.com UAT OHSS proxy
2	Allow	166.50.89.77	snfuohs01.vzbi.com UAT OHSS proxy
3	Allow	166.46.163.188	omjngohs01.vzbi.com OHSS proxy
4	Allow	166.50.88.5	snfngohs01.vzbi.com OHSS proxy
5	Allow	166.50.29.189	elmngohs01.vzbi.com OHSS proxy
6	Allow	166.35.65.2	astngohs01.vzbi.com OHSS proxy
7	Allow	152.189.67.133	sajc3p1ohs01v OHSS proxy
8	Allow	152.189.67.181	sajc3u1ohs01v UAT OHSS proxy
9	Allow	152.189.74.5	svbc3p1ohs01v OHSS proxy
10	Allow	152.189.74.53	svbc3u1ohs01v UAT OHSS proxy
default	Deny	*	The default action.

Rows per page: 50 1 – 11 of 11

1.2. Functional Sequence

1.2.1. Voice AI network function connects to a Dialogflow Connector using Websockets

- The Voice AI network function callplan loads the Voice AI network function with the Voice AI Connector WSS URI set to the Voice AI Connector for that customer project.
- The Voice AI network function connects to the Dialogflow Connector using the Database SIBB to create a new websocket connection.
- The Voice AI network function sends a CallOffered websocket message to the Voice AI Connector.

1.2.2. Dialogflow Connector requests Speech-to-Text recognition

- The Dialogflow Connector sends a CreateSession mediaRelay message to Voice AI network function.
- The Voice AI network function automatically identifies the CreateSession message and proceeds to create the media relay session.
- The Voice AI network function automatically relays all media relay messages from the media relay to the Dialogflow Connector websocket. The Dialogflow Connector receives the CreateSession response.

1.2.3. Dialogflow Connector Connects to Dialogflow

- The Dialogflow Connector connects to dialogflow.cloud.google.com using the appropriate credentials.

- The Dialogflow Connector creates a new Dialogflow conversation with the Dialogflow API.
- The Dialogflow API sends initial prompt text with utterance audio generated by Google Text-to-Speech.
- The Dialogflow Connector constructs and sends a media relay Record message. The Voice AI network function relays this message to the media relay, which arms the voice activity detector.
- The Dialogflow Connector constructs and sends one or more media relay Play message with the audio received from the Dialogflow API.
- The Voice AI network function forwards the Play messages to the media relay, which queues each block of audio and begins audio playback.

1.2.4. Dialogflow Connector Receives Recognition Results

- The media relay voice activity detector detects voice and begins streaming AudioData messages with the caller utterance.
- The Voice AI network function continuously relays AudioData messages to the Dialogflow Connector.
- The Dialogflow Connector forwards the recorded audio to Dialogflow speech-to-text streaming API.
- The media relay voice activity detector detects end-of-speech and sends an EndOfSpeech message.
- The Dialogflow Connector receives the EndOfSpeech and closes the Dialogflow API voice stream.
- The Dialogflow API processes the streamed utterance, and responds with an Intent object possibly including additional text-to-speech prompting, repeating the process.

1.2.5. Dialogflow determines intent and returns results

- The Dialogflow API returns a final Intent object with all collected intent and data elements populated.
- The Dialogflow Connector handles the results, populating an Exit message with the Intent and parameters.
- The Voice AI network function exits, closing the media relay and Dialogflow Connector websockets.
- The Voice AI Connector parses the returned results and executes business logic based on them.

1.3. Google Dialogflow CX Connector Low-Level Design

This service is a websocket server that implements the Voice AI websocket protocol to control Voice AI network function media relay resources and a Google Dialogflow CX client.

The sequence of integration follows this general flow:

1.3.1. Voice AI network function connects to this Dialogflow Voice AI Connector using Websockets

- The Voice AI Connector-enabled callplan initializes the Voice AI network function with the WSS URI set to the cloud-hosted Dialogflow connector service. The URI can have URI parameters "agent", "language", or "session" to override defaults from config.json. Any other parameters will be passed as parameters to the Dialogflow session.
- The Voice AI network function connects to the Dialogflow connector using the Database SIBB to create a new websocket connection.
- The Voice AI network function sends a CallOffered websocket message to the connector.
- The connector parses the CallOffered message and parses fields from the WSS URI to override agent, language, and session parameters for the Dialogflow session, if provided

```
    if(parsed_message.method === 'CallOffered')
  {
    // start with defaults
    session = parsed_message;
    session.digitBuffer = "";
    session.callId = parsed_message.ani.substring(6,10);
    session.dialogflowSessionId = uuid.v4();
    session.projectId = config.projectId;
    session.location = config.location;
    session.agentId = config.agentId;
    session.languageCode = config.languageCode;
    connection.callId = session.callId;

    session.parameters = { fields: { } };
    requestURL.searchParams.forEach((value, name, searchParams) => {
      // override based on the URI params
      if(name === "agent")
        session.agentId = value;
      else if(name === "language")
        session.languageCode = value;
      else if(name === "session")
        session.dialogflowSessionId = value;
      else
        session.parameters.fields[name] = { kind: 'stringValue',
stringValue: value };

    });
    logMessaging(session.callId,"FROM VERIZON
IPIVR",message.utf8Data);
    onCallOffered(session,connection);
  }
```

1.3.2. Dialogflow Voice AI Connector requests a media relay session for recording and audio playback

The Voice AI network function sends a CreateSession message to Voice AI Connector for Dialogflow on the websocket connection:

```
function sendCreateSession(session,connection) {
  var CreateSession =
    { "method":"CreateSession",
      "type":"mediaRelay",
      "version":1.0,
      "sessionId": uuid.v4(),
      "requestId": uuid.v4()
    };
  logMessaging(session.callId,"TO VERIZON
IPIVR",JSON.stringify(CreateSession));
  connection.sendUTF(JSON.stringify(CreateSession));
}
```

The Voice AI network function automatically identifies the CreateSession message and proceeds to create the media relay session.
The Voice AI network function automatically relays all media relay messages from the media relay to the Voice AI Connector websocket. The Voice AI Connector receives the CreateSession response.

1.3.3. Voice AI Connector for Dialogflow Connects to DialogFlow

- This Voice AI Connector for Dialogflow connects to dialogflow.cloud.google.com using the appropriate credentials and creates a new Dialogflow session using the Dialogflow CX API

```
   const request = {
      session: session.dialogflowSessionPath,
      queryInput: {
        // welcome intent
        intent: { intent: "projects/" + session.projectId +
                          "/locations/"+session.location +
                          "/agents/" + session.agentId +
                          "/intents/00000000-0000-0000-0000-000000000000" },
        languageCode: session.languageCode
      },
      queryParams: {
        "payload": {
          "fields": {
            "telephony.caller_id": { kind: 'stringValue', stringValue:
session.ani }
          }
        },
        "parameters": session.parameters
      }
    };
  dialogflowDetectIntent(request,session,connection);
```

- The Dialogflow API returns a welcome intent including a text prompt and rendered TTS audio for Voice AI network function to play

1.3.4. Voice AI Connector for Dialogflow initiates recognition

- This Voice AI Connector sends Play message to Voice AI network function with prompt audio, in 80kB chunks.

```
function sendPlay(audio,session,connection) {
  var start = 58;
  var length = audio.length;
  while(start < length) {
    var audioBuffer = audio.slice(start,start+80000);
    var Play = {
      "method":"Play",
      "version":"1.0",
      "sessionId":session.sessionId,
      "requestId":uuid.v4(),
      "audioData": audioBuffer.toString('base64'),
      "bargeIn": true
    };
    start += 80000;
    connection.sendUTF(JSON.stringify(Play));
    delete Play.audioData;
    logMessaging(session.callId,"TO VERIZON IPIVR",JSON.stringify(Play));
  }
}
```

- The media relay queues the prompt audio for immediate playback.
- The audio play completes, and the Voice AI network function sends a PlayComplete
- This Voice AI Connector for Dialogflow constructs and sends a media relay Record message. The Voice AI network function relays this message to the media relay.

```
async function recognize(session,connection) {
  var Record = {
      "method":"Record",
      "version":"1.0",
      "sessionId":session.sessionId,
      "requestId":uuid.v4(),
      "speechDetectionSensitivity": config.speechDetectionSensitivity,
      "utteranceEndSilence": config.utteranceEndSilence
  };
  session.digitBuffer = "";
  session.dialogflowStream = await dialogflowClient.streamingDetectIntent();
  session.dialogflowStream.on('data', response => {
      if(response.hasOwnProperty("detectIntentResponse")) {

setTimeout(onDialogflowIntent,config.dialogDelay,response.detectIntentResponse
,session,connection);
      }
      else if(response.recognitionResult != null &&
              response.recognitionResult.messageType === "TRANSCRIPT" &&
              response.recognitionResult.isFinal === true) {
```

```
          logSpeech(session.callId,' -->
'+response.recognitionResult.transcript);
      }
      else if(response.recognitionResult != null &&
response.recognitionResult.messageType === "END_OF_SINGLE_UTTERANCE") {
        stopStop(session,connection);
      }
    });
  session.dialogflowStream.on('error', err => {
      logDebug('Dialogflow Error: '+err);
    });
  session.dialogflowStream.on('end', () => {
      logDebug('Dialogflow Stream End.');
    });
  const streamRequest = {
       session: session.dialogflowSessionPath,
       queryInput: {
            audio: {
              config: {
                audioEncoding: 'AUDIO_ENCODING_MULAW',
                sampleRateHertz: 8000,
                singleUtterance: false
              }
            },
            languageCode: session.languageCode,
          },
      outputAudioConfig: {
        audioEncoding: 'OUTPUT_AUDIO_ENCODING_MULAW',
        sampleRateHertz: 8000,
        synthesizeSpeechConfig: {
          voice: {
            name: config.ttsVoice[session.languageCode],
            ssmlGender: config.ssmlGender
          }
        }
      }
    };
  session.dialogflowStream.write(streamRequest);

  logMessaging(session.callId,'TO VERIZON IPIVR',JSON.stringify(Record));
  connection.sendUTF(JSON.stringify(Record));
  session.recordActive = true;
}
```

1.3.5. Voice AI Connector for Dialogflow Receives Caller Audio

- The Voice AI Connector for Dialogflow receives audio from the media relay and streams it to Google Dialogflow

```
function onAudioData(message,session,connection) {
  try {
    if(session.hasOwnProperty("noInputTimer")) {
```

```
      clearTimeout(session.noInputTimer);
      delete session.noInputTimer;
      clearTimeout(session.interDigitTimer);
      delete session.interDigitTimer;
    }
    if(session.hasOwnProperty("dialogflowStream")) {
      const streamRequest = {
          session: session.dialogflowSessionPath,
          queryInput: {
            audio: { audio: Buffer.from(message.audioData,'base64') },
            languageCode: session.languageCode
          }
      };
      session.dialogflowStream.write(streamRequest);
      if(message.hasOwnProperty("state") && message.state == "complete") {
        session.recordActive = false;
        session.dialogflowStream.end();
      }
    }
  } catch(err) {
      console.log(timestamp() + ' ' + err);
  }
}
```

- Dialogflow performs speech-to-text recognition and reports the results
- The Dialogflow API responds with an additional Intent object possibly including audio prompting

1.3.6. DialogFlow determines final intent and exits

The Dialogflow returns a result object with all collected intent and data elements populated and endInteraction set to true.

```
function onDialogflowIntent(message,session,connection) {
  session.bargeIn = config.bargeIn;
  clearTimeout(session.noInputTimer);
  delete session.noInputTimer;
  message.queryResult.responseMessages.forEach(function(response) {
    if(response.message === "endInteraction") {
      session.state = "End";
      session.intent = message.queryResult.match;
      session.intent.session = session.dialogflowSessionId;
    } else if(response.message === "liveAgentHandoff") {
      session.state = "End";
      session.intent = message.queryResult.match;
      session.intent.liveAgentHandoff = response.liveAgentHandoff;
      session.intent.parameters = message.queryResult.parameters;
      session.intent.session = session.dialogflowSessionId;
    }
...
    promptAndExit(session,connection);
```

```
}
```

- The Voice AI Connector for Dialogflow plays final prompt audio, if provided, and then sends an Exit request with the final Intent to the Voice AI network function.

```
function promptAndExit(session,connection) {
  if(session.hasOwnProperty("prompt")) {
    sendPlay(session.prompt,session,connection);
  } else sendExit(session.intent,session,connection);
}
```

2. Azure Bot with Azure Speech Synthesis and Azure Recognizer

https://www.verizon.com/business/supportfiles/voice_ai_connector_examples_2025-09-02.tar

2.1. Solution Summary

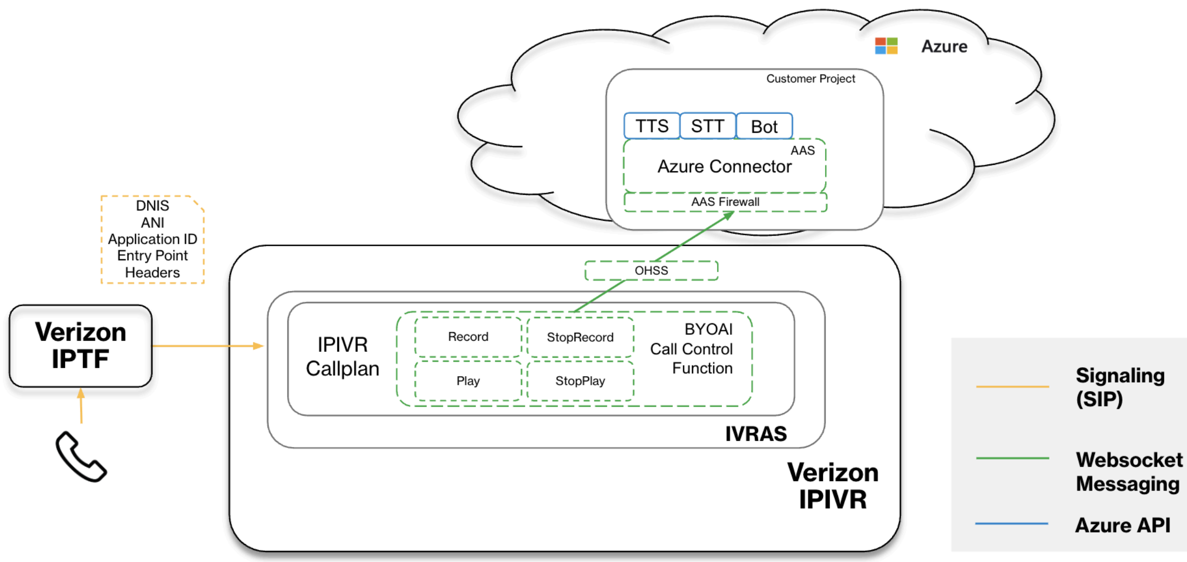
The example integration with Azure Bot uses three Azure APIs in parallel, and coordinates their asynchronous messaging with the Record and Play websocket APIs of the Voice AI Connector call control function.

Specifically, when the call is offered to the connector, the Voice AI Connector for Azure Bot creates a new conversation with the configured Azure Bot. This API will send prompt messages asynchronously, but will identify recognition turns by the inputHint parameter of the Azure Bot DirectLine API “message” activity. The prompt messages are then sent to the Azure Text-To-Speech (TTS) API to render the prompt as audio. As the audio response is streamed from the TTS API, it is played on the Voice AI network function using the Play websocket message.

When the inputHint is present and is set to “expectingInput” then the connector will arm the Record function of the Voice AI network function. When a caller utterance is detected, the Record function will begin to stream recorded audio to the connector. The connector will then stream the recorded audio to the Azure Speech-To-Text API (STT) in order to perform recognition. When the Azure STT API returns a recognition result, it is sent to the Azure Bot as a “replyToId” message activity type on the DirectLine interface.

Please note that asynchronous chat bots (that is, not directed-dialog IVRs bots) might be designed to trigger on “acceptingInput” instead, in which case the example can be modified to trigger recording and recognition asynchronously on that inputHint instead of “expectingInput” (or both).

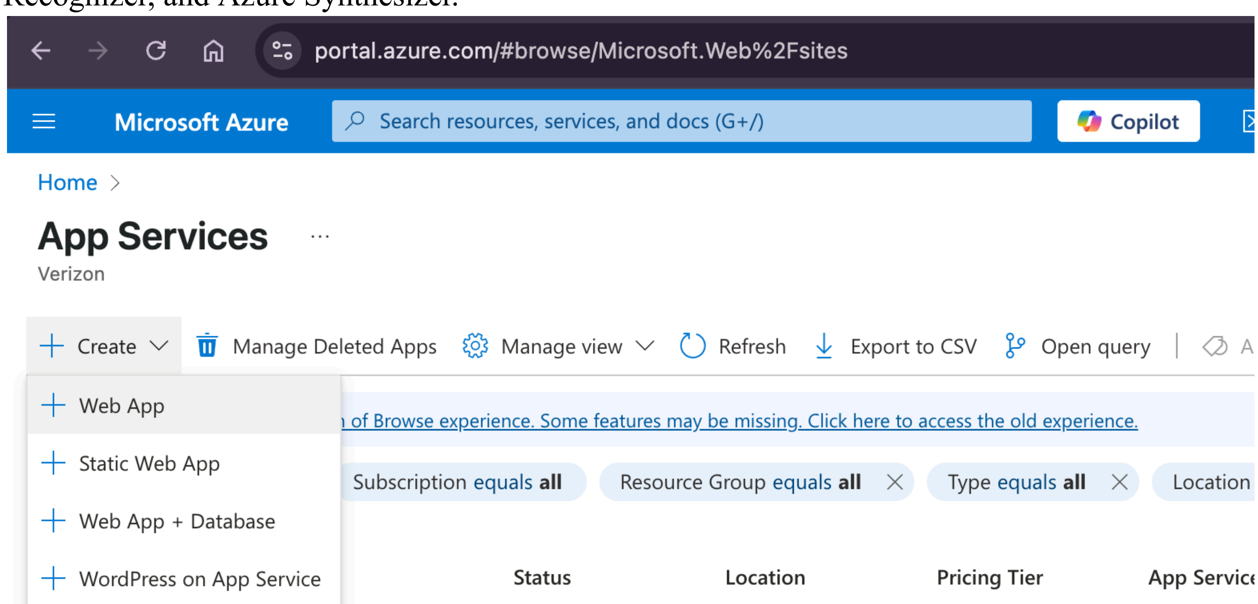
This continues until the Azure Bot indicates an end-of-conversation with a prompt containing METADATA[method=”hangup”]. This hangup metadata can pass additional name-value pairs to the IPIVR platform for call control logic after the connector ends, e.g METADATA[method="hangup" transfer="+18330220220" language="es-ES" agentSelection="payment_processing"].



2.2. Deployment Procedures

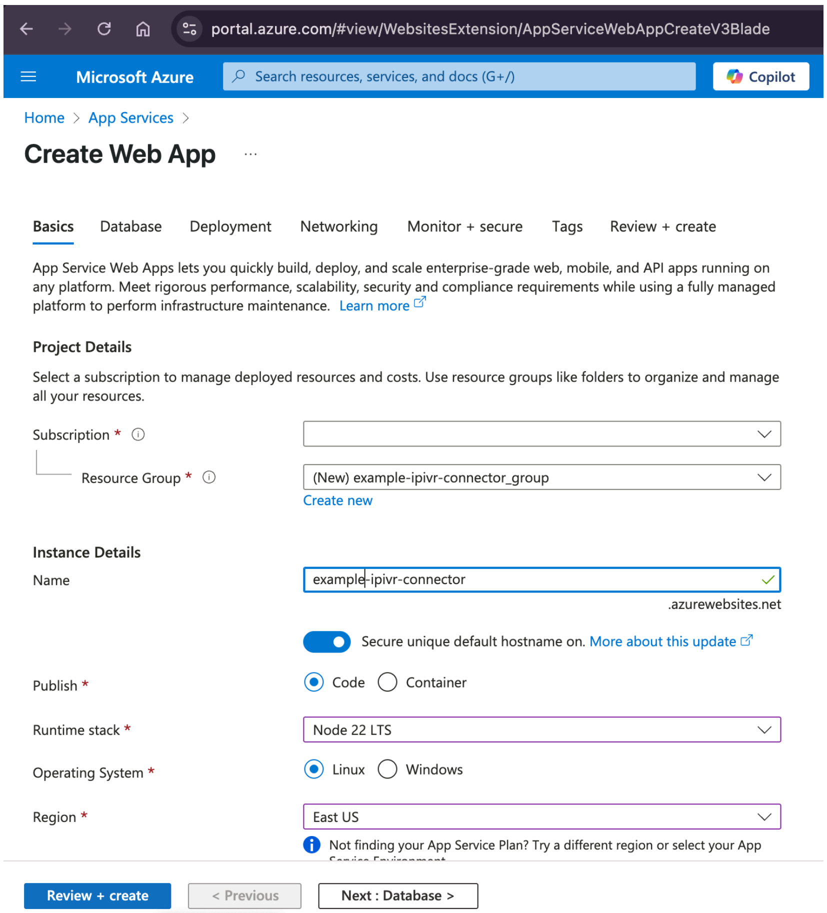
2.2.1. Create new Azure App Service Web App

This Web App will host the Voice AI Connector for Azure. It will provide communications between the Voice AI network function and three APIs provided by Azure: Azure Bot, Azure Recognizer, and Azure Synthesizer.

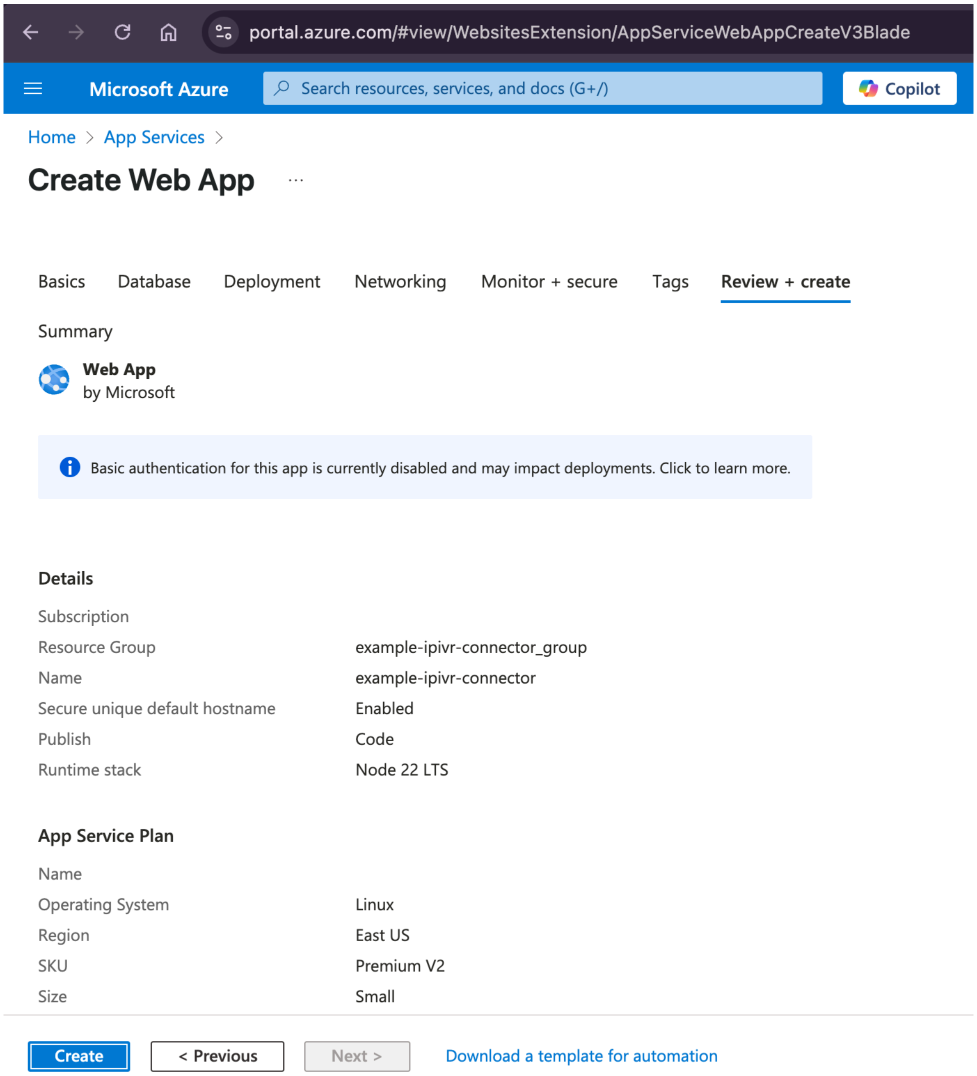


Choose Create + Web App on the Azure portal for App Services. Name the new web service and configure it as follows:

Publish: Code
Runtime stack: Node 22 LTS
Operating System: Linux
Region: *customer preference per Azure App Service Plan*
Hostname: secure unique default hostname



This will create an empty web application that can run the Voice AI Connector for Azure Bot logic.

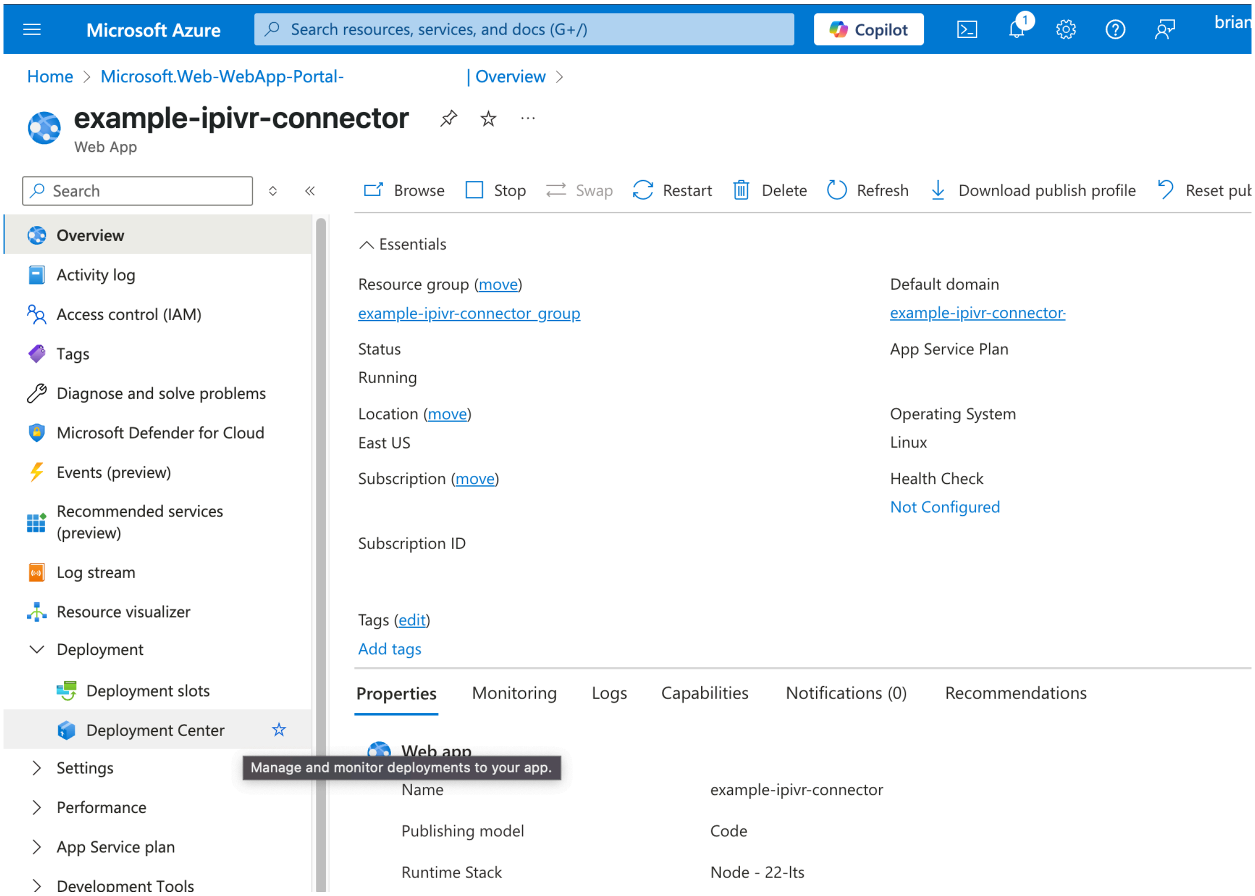


2.2.2. Create a deployment in Azure App Service to deploy the connector to the web application

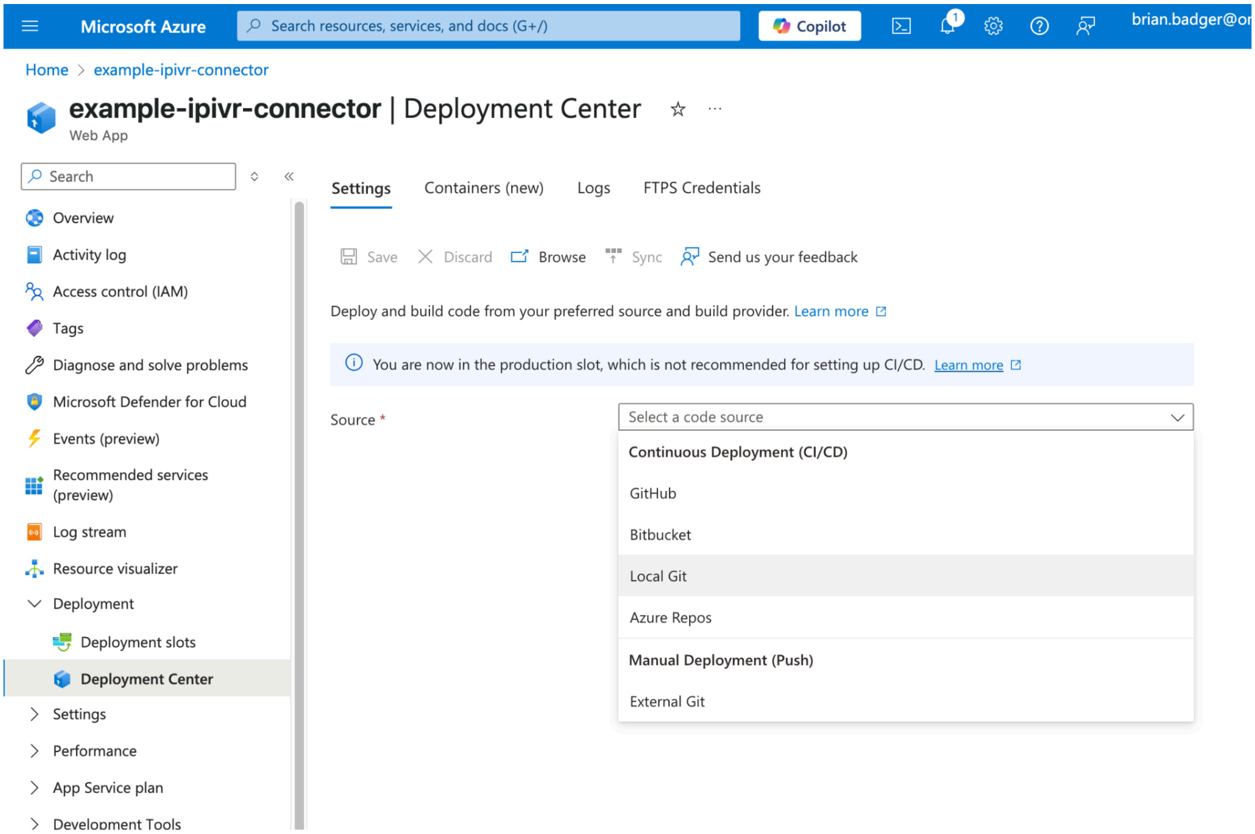
In this example deployment, we will use the Local Git feature of Azure App Service. Azure will host a Git repository for this application, and we will be able to upload our custom connector logic to the service by using git operations such as clone, add, commit and push.

Azure will automatically deploy the connector as a web app whenever we push changes to the repository. This is the simplest model if you do not already have an Azure deployment strategy in place as part of a larger web application ecosystem.

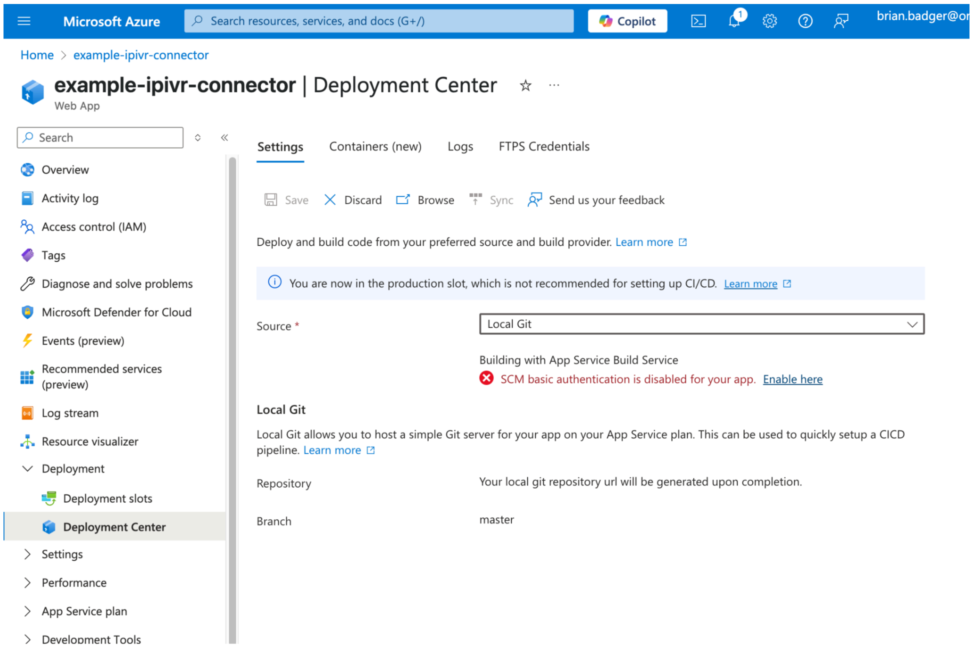
To get started, on the App Service sidebar expand Deployment and choose Deployment Center.



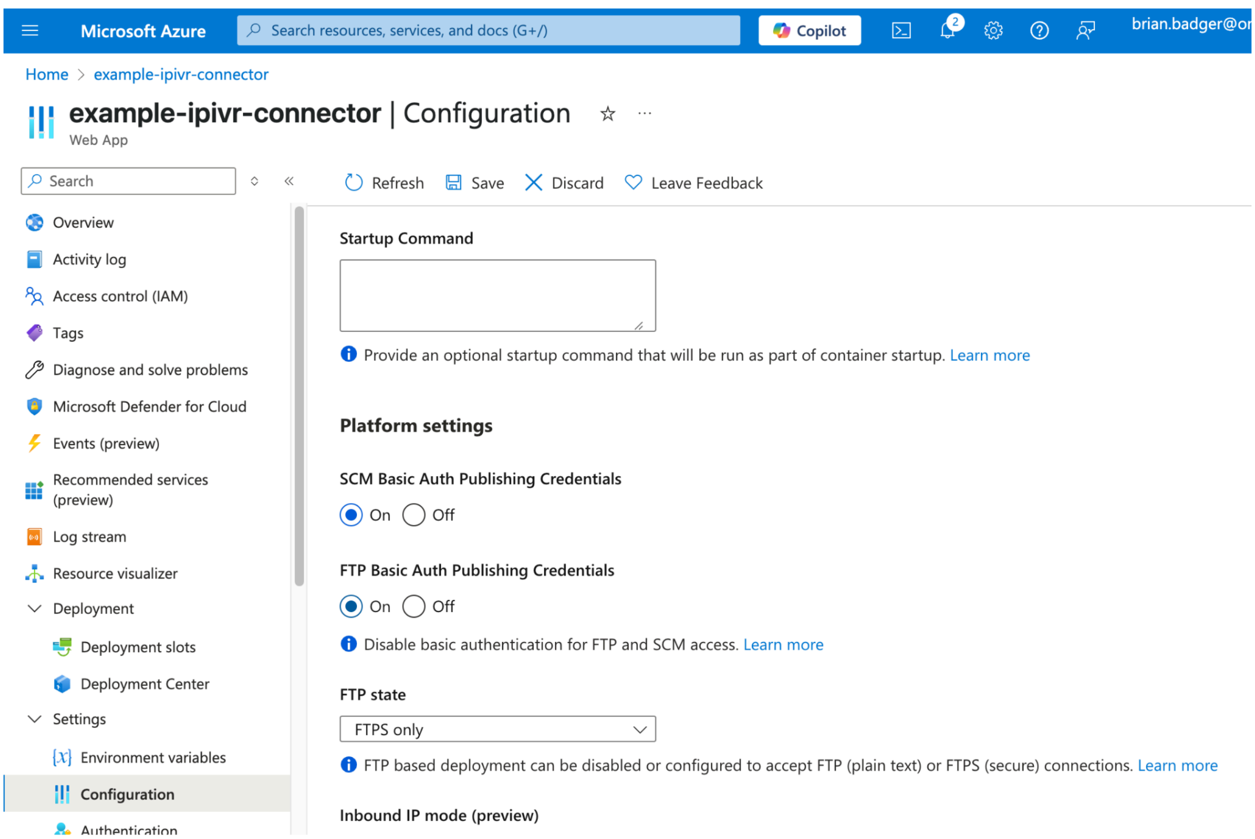
On the Settings tab, choose Source "Local Git".



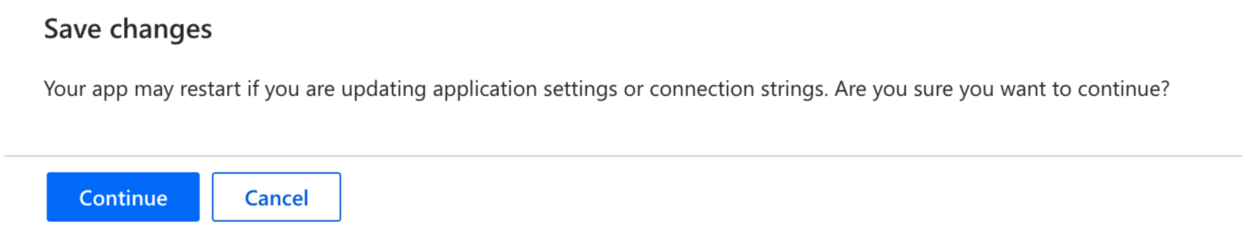
At this point, if you do not already have access enabled as part of a larger application deployment, it will require you to establish SCM basic credentials in order to authenticate with the Local Git repository. Click the "enable here" link if so required which will take you to the Settings -> Configuration sidebar tab:



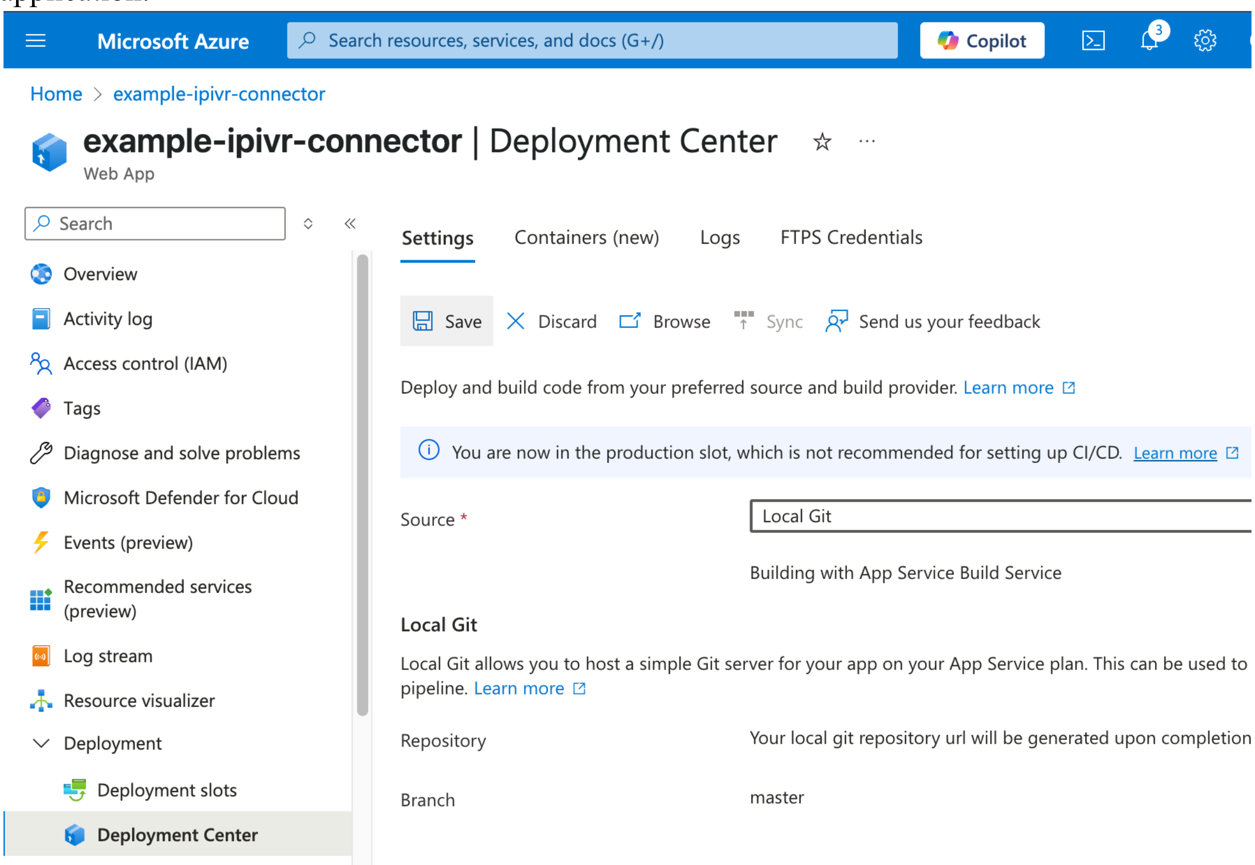
On the Settings -> Configuration tab, enable both SCM Basic Auth Publishing Credentials and FTP Basic Auth Publishing Credentials.



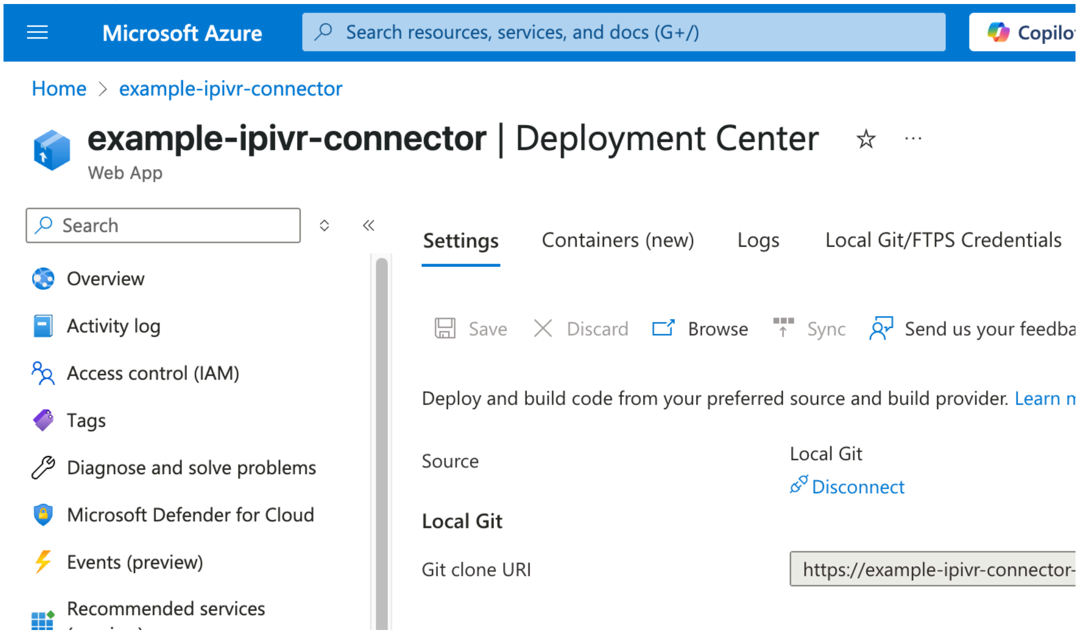
Click save and confirm:



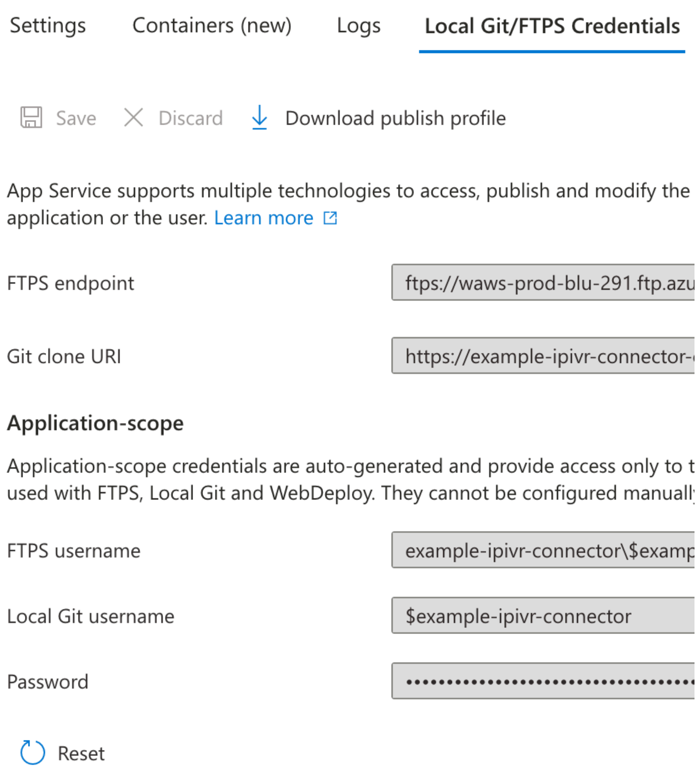
With authentication enabled, Azure should now allow Local Git to be selected as a Deployment Source. Click Save on the Deployment Center and a new git repository will be created for your application:



After saving, the Deployment Center will display the Git clone URI for your deployment repository:



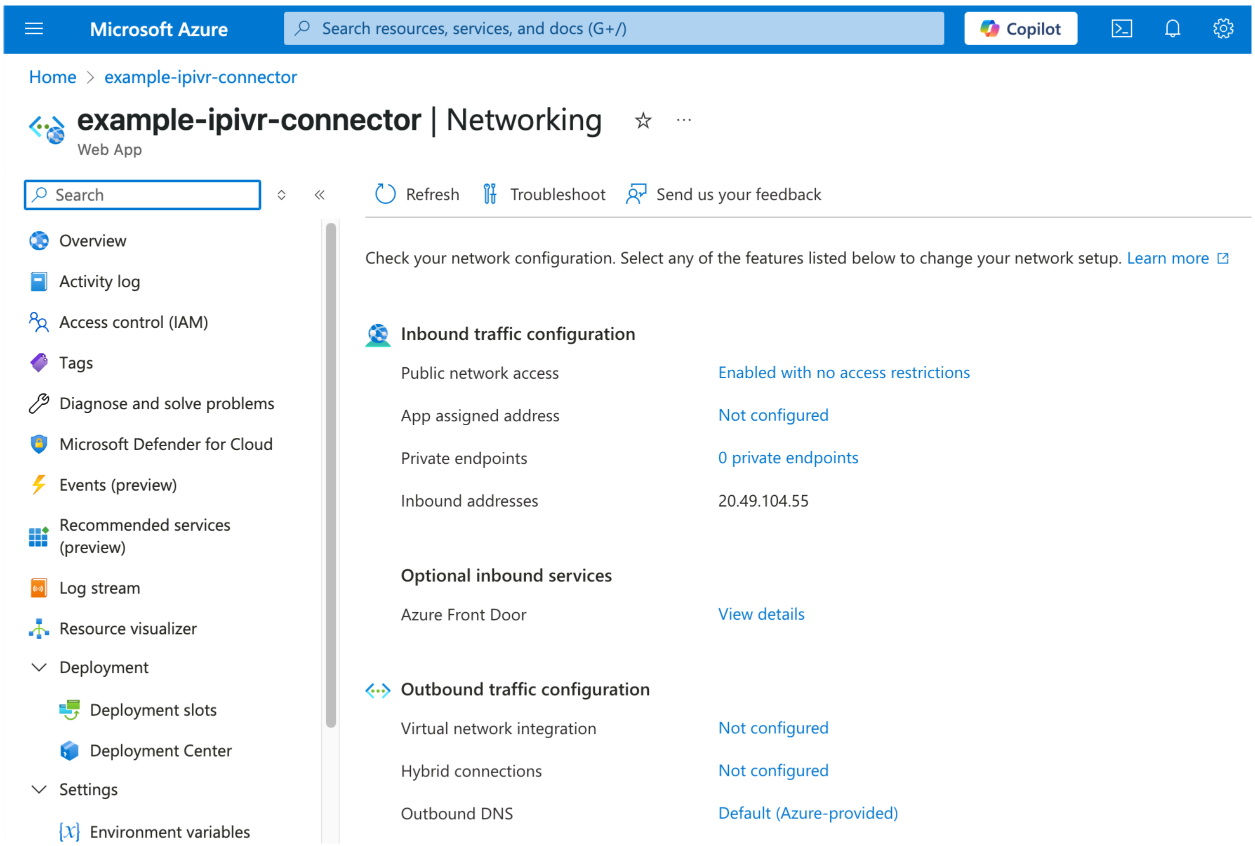
Next, we need to find our git credentials on the Local Git/FTPS Credentials tab. You will copy and use both the Local Git username and password whenever you clone or push to the repository.



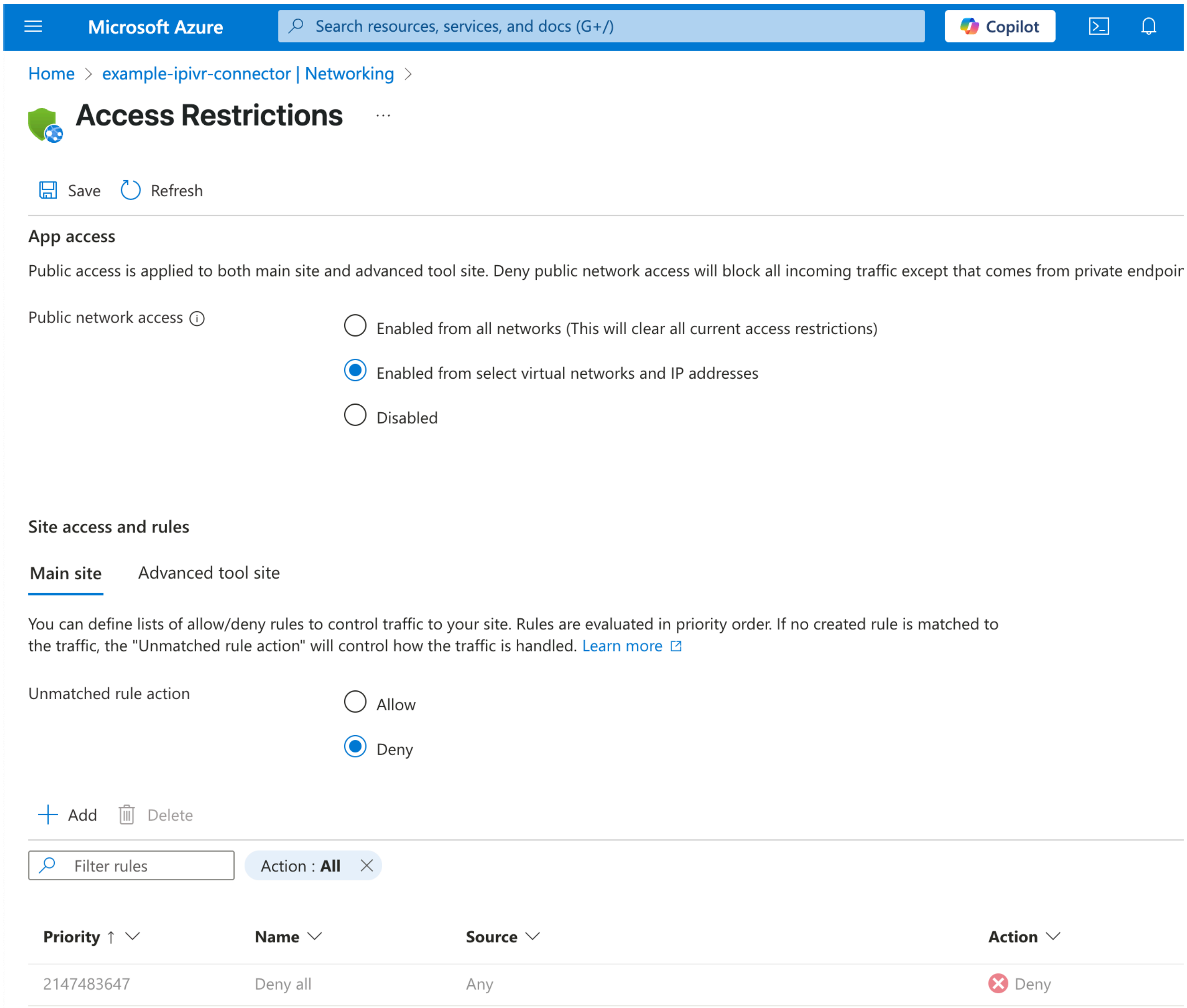
2.2.3. Enable App Service Firewall

Before deploying the connector, the Azure App Service firewall should be enabled and only select OHSS server IP addresses allowed to access the service.

Select Settings->Networking from the sidebar and click on the current value for “Public network access” to change the “Enabled with no access restrictions” default, to enable firewall restrictions:



Change the “Public network access” setting to “Enabled with select virtual networks and IP addresses” and an “Unmatched rule action” of “deny”. This will block all inbound traffic.



Finally, add each OHSS server IP address to the filter rules to enable access from Verizon to the connector.

+ Add   Delete

Filter rules   Action : **All**

Priority ↑	Name	Source	Action	HTTP head...
1	astuohs01.vzbi.com OH...	166.35.66.78/32	Allow	Not configured
2	snfuohs01.vzbi.com OH...	166.50.89.77/32	Allow	Not configured
3	omjngohs01.vzbi.com O...	166.46.163.188/32	Allow	Not configured
4	snfngohs01.vzbi.com O...	166.50.88.5/32	Allow	Not configured
5	elmngohs01.vzbi.com O...	166.50.29.189/32	Allow	Not configured
6	astngohs01.vzbi.com O...	166.35.65.2/32	Allow	Not configured
7	ndcdvohs01v.vzbi.com ...	166.40.100.100/32	Allow	Not configured
8	pdcdvohs01v.vzbi.com ...	165.122.81.100/32	Allow	Not configured
9	sajc3p1ohs01v.21sip.com	152.189.67.133/32	Allow	Not configured
10	sajc3u1ohs01v.21sip.co...	152.189.67.181/32	Allow	Not configured
11	svbc3p1ohs01v.21sip.co...	152.189.74.5/32	Allow	Not configured
12	svbc3u1ohs01v.21sip.co...	152.189.74.53/32	Allow	Not configured
2147483647	Deny all	Any	Deny	Not configured

For reference, the OHSS IP addresses are:

166.35.66.78
166.50.89.77
166.46.163.188
166.50.88.5
166.50.29.189
166.35.65.2
166.40.100.100
165.122.81.100
152.189.67.133
152.189.67.181
152.189.74.5
152.189.74.53

2.2.4. Perform the first deployment

First, copy the Git clone URI to the clipboard and clone the empty repository to your workstation, and then unpack the example connector to the directory:

```
~$ git clone
https://example-ipivr-connector-xxxxxxxxxxxxxx.scm.eastus-01.azurewebsites.net:443/example-ipivr-connector.git
~$ cd example-ipivr-connector
~/example-ipivr-connector$ tar xzvf ~/Downloads/verizon-connector-azure-bot-r25.03.01.tar.gz
x azure-bot.js
x config.json
```

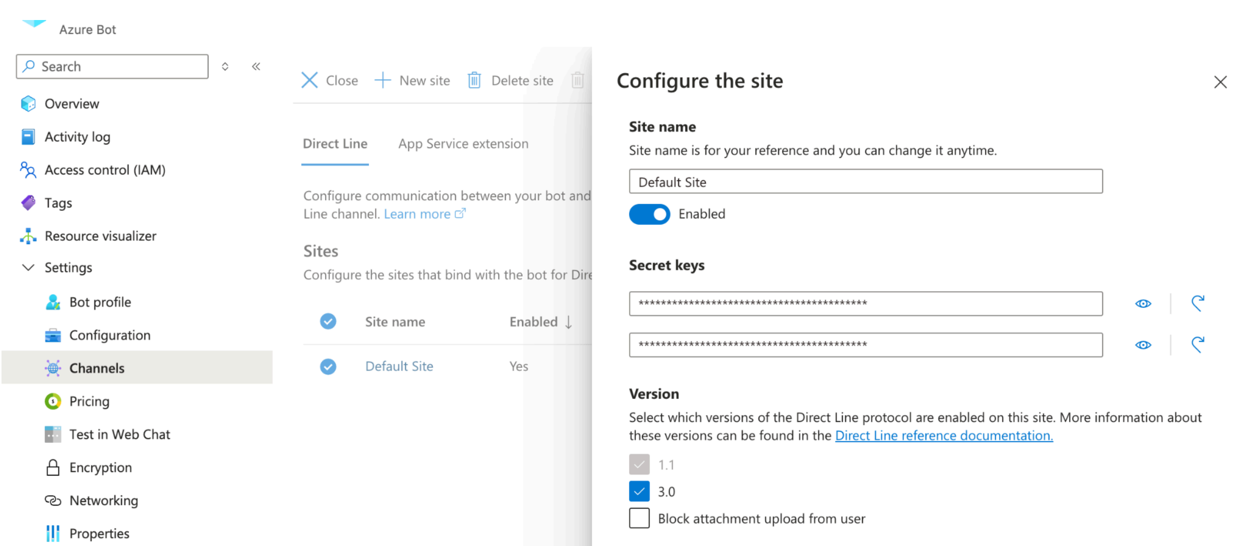
```
x package.json
x README.md
x start.js
```

Next, modify the config.json file to add credentials to access the Azure speech APIs:

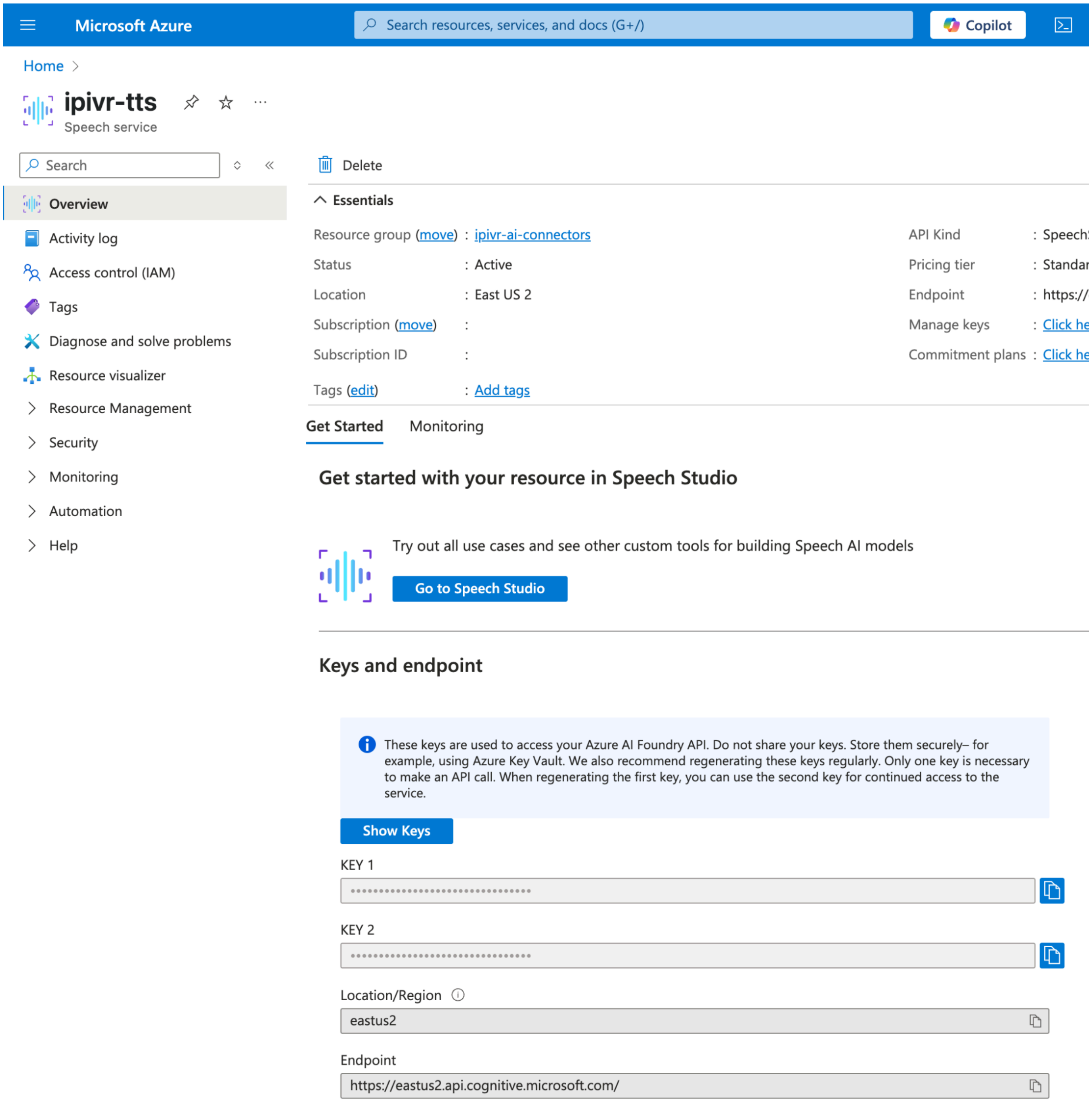
```
~/example-ipivr-connector$ vim config.json
```

```
{
  "dialogDelay": 100,
  "noInputTimeoutMs": 120000,
  "interDigitTimeoutMs": 3000,
  "speechDetectionSensitivity": 300,
  "utteranceEndSilence": 2000,
  "botSecret": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
  "synthesisSubscriptionKey":"yyyyyyyyyyyyyyyyyyyyyyyyyyy",
  "synthesisServiceRegion":"eastus",
  "synthesisLanguage":"en-US",
  "recognitionSubscriptionKey":"zzzzzzzzzzzzzzzzzzzzzzzzzzz",
  "recognitionServiceRegion":"eastus",
  "recognitionLanguage":"en-US",
  "ttsVoice": { "en-US": "en-US-AvaMultilingualNeural" },
  "languageCode": "en-US",
  "bargeIn": true,
  "hangupDelay": 1000,
  "logSpeech": true,
  "logMessaging": true,
  "logDebug": true
}
```

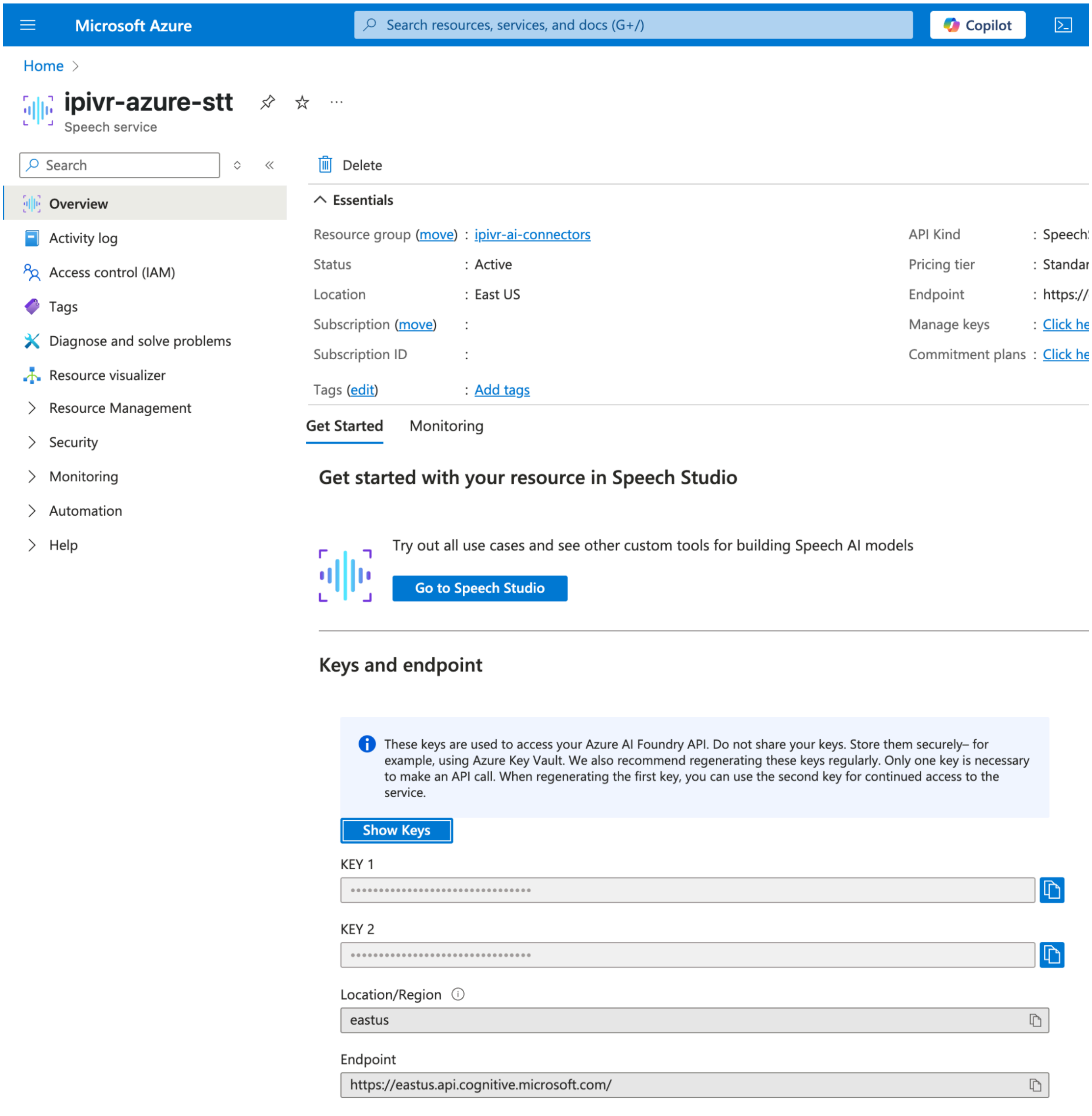
The configuration parameter "botSecret" can be found on the Azure Bot configuration page, Settings->Channels for DirectLine. View one of the "Secret keys" into the botSecret field:



The configuration parameter "synthesisSubscriptionKey" can be found on the Azure TTS configuration page, Overview. Copy one of the keys and copy the value into the "synthesisSubscriptionKey" configuration parameter value. You can also select the "ttsVoice" parameters for each language from the Azure Speech Studio Voice Gallery.



Next, the "recognitionSubscriptionKey" value can be set from the STT speech service overview. Copy the value of a key to the configuration parameter "recognitionSubscriptionKey" value.



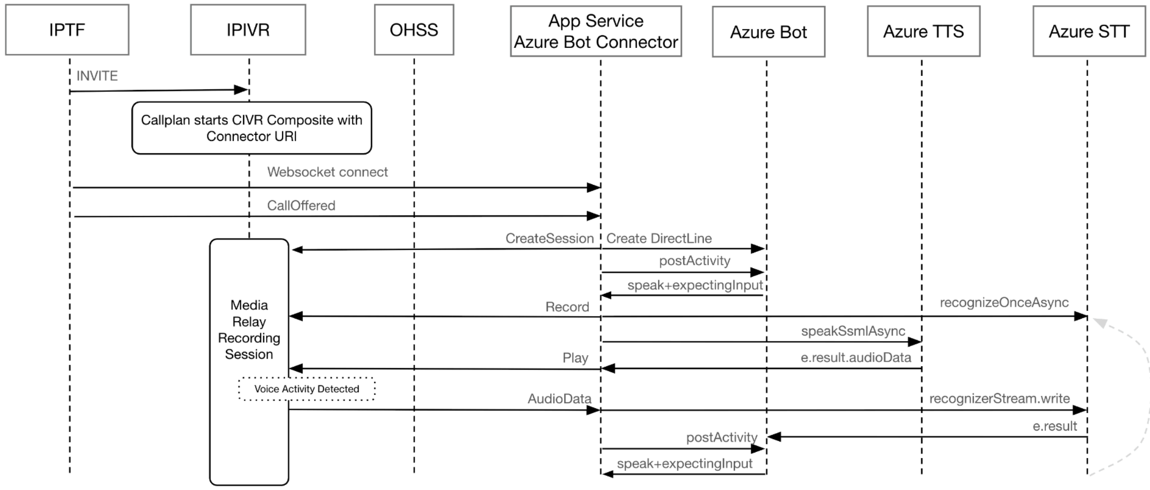
Finally, add the connector and configuration files to the repository and git push the files to Azure. Azure expects the deployment branch to be a “master” branch so be sure to set that. You will see the creation of a container image and creation of a container as part of the logging in the push operation:

```
~/example-ipivr-connector$ git checkout master
~/example-ipivr-connector$ git add azure-bot.js start.js config.json package.json
~/example-ipivr-connector$ git config --global user.email "you@example.com"
~/example-ipivr-connector$ git config --global user.name "Your Name"
~/example-ipivr-connector$ git commit -m "Initial commit"
```

```
~/example-ipivr-connector$ git push origin main:master
Username for
'https://example-ipivr-connector-xxxxxxxxxxxxxx.scm.eastus-01.azurewebsites.net:443':
$example-ipivr-connector
Password for
'https://%24example-ipivr-connector@example-ipivr-connector-xxxxxxxxxxxxxx.scm.eastus-01.azure
websites.net:443':
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 2 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 6.06 KiB | 1.01 MiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
remote: Deploy Async
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id 'f9bbe735f5'.
remote: PreDeployment: context.CleanOutputPath False
remote: PreDeployment: context.OutputPath /home/site/wwwroot
remote: Repository path is /home/site/repository
remote: Running oryx build...
remote: Operation performed by Microsoft Oryx, https://github.com/Microsoft/Oryx
remote: You can report issues at https://github.com/Microsoft/Oryx/issues
remote:
remote: Oryx Version: 0.2.20250522.1+cada9e85564f034d18420f8b5b38b3cf2259f321, Commit:
cada9e85564f034d18420f8b5b38b3cf2259f321, ReleaseTagName: 20250522.1
remote:
remote: Build Operation ID: 329c0522589c59f2
remote: Repository Commit : yyyyyyyyyyyyyyyyyyyyyyyyyyyy
remote: OS Type           : bookworm
remote: Image Type        : githubactions
remote:
remote: Primary SDK Storage URL: https://oryxsdks-cdn.azureedge.net
remote: Backup SDK Storage URL: https://oryx-cdn.microsoft.io
remote: Detecting platforms...
remote: Detected following platforms:
remote:   nodejs: 22.15.0
remote:   hugo: 0.124.1
remote: Version '22.15.0' of platform 'nodejs' is not installed. Generating script to install
it...
remote:
remote: Using intermediate directory '/tmp/8ddacda451232cc'.
remote:
remote: Copying files to the intermediate directory...
remote: Done in 0 sec(s).
remote:
remote: Source directory     : /tmp/8ddacda451232cc
remote: Destination directory: /home/site/wwwroot
remote:
remote:
remote: Downloading and extracting 'nodejs' version '22.15.0' to
'/tmp/oryx/platforms/nodejs/22.15.0'...
remote: Detected image debian flavor: bookworm.
remote: Downloaded in 0 sec(s).
remote: Verifying checksum...
remote: Extracting contents...
remote: performing sha512 checksum for: nodejs...
remote: Done in 4 sec(s).
remote:
remote: Removing existing manifest file
remote: Creating directory for command manifest file if it does not exist
remote: Creating a manifest file...
remote: Node Build Command Manifest file created.
remote:
remote: Using Node version:
remote: v22.15.0
remote:
remote: Using Npm version:
remote: 10.9.2
remote:
remote: Running 'npm install'...
```

```
remote:
remote: .......
remote: npm warn deprecated core-js@3.15.2: core-js@<3.23.3 is no longer maintained
and not recommended for usage due to the number of issues. Because of the V8 engine
whims, feature detection in old core-js versions could cause a slowdown up to 100x
even if nothing is polyfilled. Some versions have web compatibility issues. Please,
upgrade your dependencies to the actual version of core-js.
remote:
remote: npm notice
remote: added 57 packages, and audited 58 packages in 12s
remote: npm notice New major version of npm available! 10.9.2 -> 11.4.2
remote:
remote: npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.4.2
remote: 11 packages are looking for funding
remote: npm notice To update run: npm install -g npm@11.4.2
remote:   run `npm fund` for details
remote: npm notice
remote:
remote: found 0 vulnerabilities
remote:
remote: Zipping existing node_modules folder...
remote: Done in 3 sec(s).
remote: Preparing output...
remote:
remote: Copying files to destination directory '/home/site/wwwroot'...
remote: Done in 0 sec(s).
remote:
remote: Removing existing manifest file
remote: Creating a manifest file...
remote: Manifest file created.
remote: Copying .ostype to manifest output directory.
remote:
remote: Done in 21 sec(s).
remote: Running post deployment command(s)...
remote:
remote: Generating summary of Oryx build
remote: Parsing the build logs
remote: Found 0 issue(s)
remote:
remote: Build Summary :
remote: ===============
remote: Errors (0)
remote: Warnings (0)
remote:
remote: Triggering recycle (preview mode disabled).
remote: Deployment successful. deployer =  deploymentPath =
remote: Deployment Logs :
'https://example-ipivr-connector-xxxxxxxxxxxxxx.scm.eastus-01.azurewebsites.net/newui/
jsonviewer?view_url=/api/deployments/yyyyyyyyyyyyyyyyyyyyyyyyyyyy/log'
To
https://example-ipivr-connector-xxxxxxxxxxxxxx.scm.eastus-01.azurewebsites.net:443/exa
mple-ipivr-connector.git
 * [new branch]      master -> master
```

2.3. Functional Sequence



2.3.1. Voice AI connects to an Voice AI Connector for Azure Bot using Websockets

- The customer callplan starts the Voice AI network function with the Voice AI Connector WSS URI set to the Voice AI Connector for Azure Bot for that customer project.
- The Voice AI network function connects to the Voice AI Connector for Azure Bot using the WSS URI to create a new websocket connection.
- The Voice AI network function sends a CallOffered websocket message to the Voice AI Connector for Azure Bot.

2.3.2. Voice AI Connector for Azure Bot creates new mediaRelay session

- The Voice AI Connector for Azure Bot sends a CreateSession mediaRelay message to Voice AI network function.
- The Voice AI network function receives the CreateSession message and proceeds to create the media relay session.
- The Voice AI network function automatically relays all media relay messages from the media relay to the Voice AI Connector for Azure Bot websocket. The Voice AI Connector for Azure Bot receives the CreateSession response.

2.3.3. Voice AI Connector for Azure Bot Connects to Azure Bot

- The Voice AI Connector for Azure Bot connects to the Azure Bot API using the appropriate credentials.
- The Voice AI Connector for Azure Bot creates a new Azure Bot conversation with the Azure Bot API.
- The Azure Bot DirectLine API sends initial prompt text as a “message” activity.
- If (and only if) the message contains an “inputHint” value of “inputExpected”, the Voice AI Connector for Azure Bot constructs and sends a media relay Record message. The Voice AI network function relays this message to the media relay, which arms the voice activity detector.

2.3.4. Voice AI Connector for Azure Bot generates prompt audio using text-to-speech API

- The Voice AI Connector for Azure Bot sends the received prompt text to the Azure text-to-speech API to render as playable audio.
- The Voice AI Connector for Azure Bot constructs and sends one or more media relay Play messages with the audio received from the Azure text-to-speech API. The format of the audio sent in the Play message is base64 encoded g.711u.
- The Voice AI network function forwards the Play messages to the media relay, which queues each block of audio and begins audio playback.

2.3.5. Voice AI Connector for Azure Bot Receives Recognition Results

- The media relay voice activity detector detects voice and begins streaming AudioData messages with the caller utterance.
- The Voice AI network function continuously relays AudioData messages to the Voice AI Connector for Azure Bot.
- The Voice AI Connector for Azure Bot forwards the recorded audio to Azure Bot speech-to-text streaming API.
- The media relay voice activity detector detects end-of-speech and sends an EndOfSpeech message.
- The Voice AI Connector for Azure Bot receives the EndOfSpeech and closes the Speech-to-text API voice stream.
- The Azure speech-to-text API processes the streamed utterance, and responds with the recognition results.

2.3.6. Azure Bot processes the recognition results

- The Voice AI Connector for Azure Bot sends the text-format recognition results to the Azure Bot and receives additional prompt text
- This process continues until the prompt text contains call-control metadata to end the interaction, transfer the call or hang up, at which point the Voice AI Connector for Azure Bot sends an Exit message to the Voice AI network function which executes the call control action.

2.4. Voice AI Connector for Azure Bot Low-Level Design

This service is a websocket server that implements the Voice AI websocket protocol to control Voice AI media relay resources and coordinate the interworking with three Azure API clients: Azure Bot, Azure TTS and Azure STT.

The sequence of integration follows this general flow:

2.4.1. Voice AI connects to this Voice AI Connector for Azure Bot using Websockets

- The Voice AI callplan loads the Voice AI network function with the Voice AI Connector WSS URI set to this Voice AI Connector for Azure Bot.
- The Voice AI network function connects to the Voice AI Connector for Azure Bot to create a new websocket connection.

- The Voice AI network function sends a CallOffered websocket message to the Voice AI Connector for Azure Bot.

2.4.2. Voice AI Connector for Azure Bot requests a media relay session for recording and audio playback

The Voice AI Connector for Azure Bot sends a CreateSession message to Voice AI on the websocket

```
function sendCreateSession(session,connection) {
  var CreateSession =
    { "method":"CreateSession",
      "type":"mediaRelay",
      "version":1.0,
      "sessionId": uuid.v4(),
      "requestId": uuid.v4()
    };
  logMessaging(session.callId,"TO VERIZON
IPIVR",JSON.stringify(CreateSession));
  connection.sendUTF(JSON.stringify(CreateSession));
}
```

The Voice AI network function identifies the CreateSession message and proceeds to create the media relay session with media relay resources.
The Voice AI network function automatically relays all media relay messages from the media relay to the Voice AI AS websocket. The Voice AI AS receives the CreateSession response.

2.4.3. Voice AI Connector for Azure Bot Connects to Azure Bot

- The Voice AI Connector for Azure Bot connects to Azure Bot using the appropriate credentials and creates a new Azure Bot session using the Azure Bot DirectLine API

```
function createAzureBot(session, connection) {
  session.directLine = new DirectLine({
    secret: config.botSecret,
    timeout: 10000,
    webSocket: true,
    conversationStartProperties: { /* optional: properties to send to the bot on
conversation start */
        locale: 'en-US'
    }
  });
  session.connectionSubscription =
session.directLine.connectionStatus$.subscribe(connectionStatus => {
    logDebug("DirectLine connection status "+connectionStatus);
    if(connectionStatus == 2)
      startConversation(session,connection);
  });
```

- The Voice AI Connector for Azure Bot starts a new conversation with an initial utterance such as “hello bot”

NOTE: There are known race conditions with the DirectLine interface that may cause the onMembersAdded() event to never fire in the bot when the connection is established. So instead of triggering the bot on onMembersAdded() we recommend designing your bot to start a conversation on an explicit message like “hello bot”, as in this example

```
function startConversation(session,connection) {
  if(session.hasOwnProperty("directLine")) {
    var activity = {
      from: { id: session.sessionId },
      type: "message",
      text: "hello bot"
    };

    logMessaging(session.callId,"TO AZURE",JSON.stringify(activity));
    session.directLine.postActivity(activity).subscribe(
      id => {
      },
      error => {
        logDebug('Error posting activity to bot: '+error)
      }
    );
  }
}
```

- The Azure Bot API asynchronously sends prompt text which is queued for a recognition turn.

```
  session.activitySubscription = session.directLine.activity$.filter(activity => activity.type
=== 'message').subscribe(message => {
    logMessaging(session.callId,"FROM AZURE", JSON.stringify(message));
    if(message.hasOwnProperty("conversation") && message.conversation.hasOwnProperty("id") &&
!session.hasOwnProperty("conversationId"))
      session.conversationId = message.conversation.id;
    if(message.hasOwnProperty("speak"))
      queueTTSPrompt(message.speak,session,connection);
```

- If the prompt indicates an inputHint of “expectingInput” and bargeIn is enabled, then the queued prompt text is rendered as audio by the Azure text-to-speech API and the Voice AI Record API is armed to collect caller utterance audio:

```
    if(message.hasOwnProperty("inputHint") && message.inputHint === "expectingInput")
      {
        session.turn += 1;
        session.azureResults = "";
        session.dtmfResults = "";
        session.textPrompt = "";
        if(message.hasOwnProperty("speak")) {
          session.textPrompt = message.speak;
```

```
      }
      delete session.recognitionActivityId;
      if(message.hasOwnProperty("id")) {
        session.recognitionActivityId = message.id;
      }
      promptAndRecognize(session,connection);
    }
```

2.4.4. Voice AI Connector for Azure Bot initiates recognition

- As the Azure text-to-speech API returns binary audio, the Voice AI Connector sends Play websocket messages to Voice AI with the prompt audio, in 80kB chunks.

```
function sendPlay(session,connection) {
  var start = 0;
  var audio = session.prompt;
  delete session.prompt;
  var length = audio.length;
  while(start < length) {
    var audioBuffer = audio.slice(start,start+80000);
    var Play = {
      "method":"Play",
      "version":"1.0",
      "sessionId":session.sessionId,
      "requestId":uuid.v4(),
      "audioData": audioBuffer.toString('base64'),
      "bargeIn": true
    };
    start += 80000;
    connection.send(JSON.stringify(Play));
    delete Play.audioData;
    logMessaging(session.callId,"TO VERIZON",JSON.stringify(Play));
  }
}
```

- The media relay queues the prompt audio for immediate playback.
- The audio play completes, and the Voice AI network function sends a PlayComplete
- If bargeIn was not enabled, after PlayComplete, the Voice AI Connector for Azure Bot constructs and sends a media relay Record message and starts a recognition session with the Azure speech-to-text API.
- If bargeIn is enabled, before the Play message is sent, the Voice AI Connector for Azure Bot constructs and sends a media relay Record message and starts a recognition session with Azure speech-to-text API.

```
function recognize(session,connection) {
  if(session.recordActive)
    return;
  var Record = {
      "method":"Record",
```

```
        "version":"1.0",
        "sessionId":session.sessionId,
        "requestId":uuid.v4(),
        "speechDetectionSensitivity": config.speechDetectionSensitivity,
        "utteranceEndSilence": config.utteranceEndSilence
  };
  session.digitBuffer = "";
  var format = sdk.AudioStreamFormat.getWaveFormat(8000, 8, 1, sdk.AudioFormatTag.MuLaw);
  session.recognizerStream = sdk.AudioInputStream.createPushStream(format);
  var audioConfig = sdk.AudioConfig.fromStreamInput(session.recognizerStream);
  var speechConfig = sdk.SpeechConfig.fromSubscription(config.recognitionSubscriptionKey,
config.recognitionServiceRegion);
  speechConfig.speechRecognitionLanguage = config.recognitionLanguage;
  speechConfig.setProperty(sdk.PropertyId[sdk.PropertyId.Speech_SegmentationSilenceTimeoutMs],
"3000");
  speechConfig.outputFormat=1;
        // create the speech recognizer.
  session.recognizer = new sdk.SpeechRecognizer(speechConfig, audioConfig);
  session.recognizer.recognized = function (s, e) {
    if(e.result.reason === sdk.ResultReason.NoMatch) {
       var noMatchDetail = sdk.NoMatchDetails.fromResult(e.result);
       logSpeech(session.callId,' -[AZURE]->');
       getBotResponse("",session,connection);
    } else {
      var normalizedText = e.result.text;
      try {
        var result = JSON.parse(e.result.json);
        normalizedText = result.NBest[0].ITN;
      } catch(err) { logDebug("Error parsing recognition results: "+err); }
      logSpeech(session.callId,' -[AZURE]-> '+normalizedText);
      session.azureResults = normalizedText;
      getBotResponse(normalizedText,session,connection);
    }
  };

  // start the recognizer and wait for a result.
  session.recognizer.recognizeOnceAsync(
    function (result) {
      session.recognizer.close();
      delete session.recognizer;
      if(session.hasOwnProperty("recognizerStream")) {
        session.recognizerStream.close()
        delete session.recognizerStream;
        sendStop();
      }
    },
    function (err) {
      logDebug("Error in recognizeOnceAsync: "+err);
    });
  logMessaging(session.callId,'TO VERIZON',JSON.stringify(Record));
  connection.send(JSON.stringify(Record));
  session.recordActive = true;
}
```

- The Voice AI network function relays this message to the media relay.

2.4.5. Voice AI Connector for Azure Bot Receives Caller Audio

- The Voice AI Connector for Azure Bot receives audio from the media relay and streams it to Azure Speech To Text API

```
function onAudioData(message,session,connection) {
  try {
    if(session.hasOwnProperty("noInputTimer")) {
      clearTimeout(session.noInputTimer);
      delete session.noInputTimer;
      clearTimeout(session.interDigitTimer);
      delete session.interDigitTimer;
    }
    session.state = "Recognizing";
    if(session.hasOwnProperty("recognizerStream")) {
      session.recognizerStream.write(Buffer.from(message.audioData,'base64'));
      if(message.hasOwnProperty("state") && message.state == "complete") {
        if(session.hasOwnProperty("recognizer")) {
          session.recognizer.stopContinuousRecognitionAsync();
        }
      }
    }
    if(message.hasOwnProperty("state") && message.state == "complete") {
      session.recordActive = false;
    }
  } catch(err) {
      logDebug("Error writing to recognizer stream: "+err);
  }
}
```

- The Azure STT API performs speech-to-text recognition and returns the results
- The Voice AI Connector for Azure Bot extracts the inverse-text-normalization results of the recognition and sends this string to the Azure Bot DirectLine API as a “replyToId” “message”.
- The Azure Bot processes this input and further interactions are triggered asynchronously by the bot as above until the caller hangs up or the Bot indicates an end-of-conversation using a prompt that contains a METADATA[method=”hangup”] message.

```
function getBotResponse(text,session,connection) {

setTimeout(() => {
  if(session.hasOwnProperty("directLine")) {
    var activity = {
      from: { id: session.sessionId },
      conversation: { id: session.conversationId },
      replyToId: session.recognitionActivityId,
      type: 'message',
      text: text
    };

    logMessaging(session.callId,"TO AZURE",JSON.stringify(activity));
```

```
    session.directLine.postActivity(activity).subscribe(
      id => {
      },
      error => {
        logDebug('Error posting activity to bot: '+error)
      }
    );
  }
  },config.dialogDelay);
  }
```

- The Azure Bot processes this input and further interactions are triggered asynchronously by the bot as above until the caller hangs up or the Bot indicates an end-of-conversation using a prompt that contains a METADATA[method=”hangup”] message.

```
function parseMETADATA(input,session) {
  var output = input;
  const dataPattern = /METADATA\[(([a-zA-Z]+="[^"\\]*(?:\\.[^"\\]*)*"[,\s]*)*)\]/g;
  const paramPattern = /([a-zA-Z]+)="([^"\\]*(?:\\.[^"\\]*)*)"/g;
  const dataMatch = [...input.matchAll(dataPattern)];
  for(const match of dataMatch) {
    output = output.replace(match[0],"");
    const paramMatch = [...match[1].matchAll(paramPattern)];
    var currentData = { };
    for(const param of paramMatch) {
      currentData[param[1]] = param[2];
    }
    if(session) {
      if(currentData.hasOwnProperty("method") && currentData.method === "hangup")
      {
        session.endAfterPlay = true;
        logDebug("Received hangup message, will hangup after play completes.");
        session.intent = currentData;
      }
    }
  }
  return output;
}
```

3. Voice AI Websocket API

The Voice AI Connector product implements a websocket-based API that provides media relay functions for customer integrations with speech and natural language AI APIs. This messaging protocol is very simple, enabling a web application to record and playback raw ulaw audio over a media relay session.   This API can be used for simple media relay functions such as integration with external IVR vendors, mid-call interaction during a conference, etc. For the purposes of this analysis, it is intended to allow integration with external speech-to-text and text-to-speech web services.

For all messages the requestId and sessionId properties defined are for simplified correlation at the application. No error behavior is defined when they are missing. For all messages, the version property is informational, behavior is currently undefined if it is missing. No error text is defined.

3.1. CallOffered

Purpose: Initial message sent from Voice AI network function to the Voice AI Connector which contains network information about the call including call identifiers, calling party and called party URIs and STIR/SHAKEN identity (when allowed, subject to regulatory constraints)

<table>
<tr><th>Property</th><th>Type</th><th>Value</th></tr>
<tr><td>method</td><td>String</td><td>CallOffered</td></tr>
<tr><td>dnis</td><td>String</td><td>Dialed number, e.164 format</td></tr>
<tr><td>ani</td><td>String</td><td>Calling party number, e.164 format</td></tr>
<tr><td>media_relay_available</td><td>Bool</td><td>true if media relay functions are available (would be false if Voice AI Connector service was not enabled for the customer)</td></tr>
<tr><td>appId</td><td>String</td><td>Voice AI Connector application id</td></tr>
<tr><td>entryPoint</td><td>String</td><td>Voice AI Connector application entry point</td></tr>
<tr><td>headers</td><td>Array of object</td><td>SIP headers from initial INVITE
<table>
<tr><th>Property</th><th>Type</th><th>Value</th></tr>
<tr><td>name</td><td>String</td><td>name of header</td></tr>
<tr><td>value</td><td>String</td><td>value of header</td></tr>
</table>
</td></tr>
</table>

3.2. CreateSession

3.2.1. Request

Purpose: create a media relay websocket session with no native STT or TTS support. Media is transported over websockets.

Property	Type	Value
method	String	CreateSession
version	String	(Optional) Protocol version: 1.0
type	String	value “mediaRelay”
requestId	String	Request identifier
direction	String	Which media direction to we are creating a session for, in (from caller) or out (from agent)

Example:

{
"method":"CreateSession",
"version":"1.0",
"type":"mediaRelay",
"requestId":"d380d890-fdc6-11ea-a66b-ed36d1ad829b"
}

3.2.2. Response

Purpose: session is created, use the provided session id in all future requests

Property	Type	Value
method	String	Response
responseTo	String	CreateSession
version	String	Protocol version: 1.0
sessionId	String	Session identifier.

Property	Type	Value
requestId	String	Request identifier.

Example:

{"method":"Response",
"responseTo":"CreateSession",
"version":"1.0",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d380d890-fdc6-11ea-a66b-ed36d1ad829b",
"status":"complete"}

3.3. Record

3.3.1. Request

Purpose: start a recording

Property	Type	Value
method	String	Record
direction	String	Which media direction to record, in (from caller) or out (from agent)
version	String	1.0
sessionId	String	Session identifier
requestId	String	Request identifier.

Property	Type	Value
speechDetectionSensitivity	Int64	(Optional) Sensitivity scalar value from 0 to 1000, 0 will never detect speech (recording nothing) 1000 will detect everything (including noise and silence) as speech (recording everything) The recommended value of 500 is verified to provide good performance on normal voice sources. If not provided, the default behavior is to record all audio without voice activity detection and the BargeIn event will not fire.
utteranceEndSilence	Int64	(Optional) A duration (in milliseconds) to wait after the last detected voice sample before automatically ending the recording. Recommended value of 1000 is verified to provide good performance on normal voice sources. If not provided the recording will continue until stopped by StopRecord.
noInputTimeout	Int64	(Optional) A duration (in milliseconds) to wait for the start of speech. Default value is no timeout.
totalTimeout	Int64	(Optional) A maximum duration (in milliseconds) to record. Default is value is no timeout.
startRecognitionTimers	Int64	(Optional) If false, noInputTimeout and totalTimeout start upon receipt of StartRecognitionTimers requests, if true, timers start immediately.

Example:

{
"method":"Record",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d3819be0-fdc6-11ea-a66b-ed36d1ad829b"
}

3.3.2. Response

Purpose: notification that recording has started, AudioData events will be sent with recording audio in near real time

Property	Type	Value
method	String	Response
responseTo	String	Record
version	String	1.0
sessionId	String	Session identifier.
requestId	String	Request identifier.
errors	Array	(optional) error event text
status	String	recording

{
"method":"Response",
"responseTo":"Record",
"version":"1.0",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d3819be0-fdc6-11ea-a66b-ed36d1ad829b",
"errors":[],
"status":"recording"
}

3.3.3. Event BargeIn

Purpose: BargeIn notifies the connector that a Play in progress has stopped due to voice activity detection

Property	Type	Value
method	String	Event

Property	Type	Value
Event	String	BargeIn
direction	String	Which media direction barged, in (voice from caller) or out (voice from agent)
version	String	(Optional) Protocol version: 1.0
sessionId	String	Session identifier.
audioData	String	Base64 encoded ulaw audio of arbitrary length

Example:

{"method":"Event","event":"BargeIn","version":"1.0","sessionId":"e78ed48f-2025-445c-94f5-65aa135f41de","requestId":"dcdf8d86-def4-436e-aee8-c5b43ef87d9c"}

3.3.4. Event AudioData

Purpose: AudioData contains base64 encoded ulaw raw audio at 8khz, event sent 2.5 times per second until stopped. Application should stream this data to the speech to text API as each AudioData message is received. (Note: OpenAI Whisper currently provides only a REST API, so the audio should be accumulated in that case by the connector until recording is complete, then the REST request sent with the full recorded utterance)

Property	Type	Value
method	String	Event
event	String	AudioData
version	String	(Optional) Protocol version: 1.0
direction	String	Which media direction recorded, in (from caller) or out (from agent)
sessionId	String	Session identifier.

Property	Type	Value
audioData	String	Base64 encoded ulaw audio of arbitrary length
state	String	(Optional) value "complete" when recording end-of-speech condition is satisfied, ending the recording

Example:

{"method":"Event","event":"AudioData","version":"1.0","sessionId":"D75AB024-9DEC-4F0B-A675-E94883717170","audioData":"/w=="}

3.4. StopRecord

3.4.1. Request

Purpose: stop a recording

Property	Type	Value
method	String	StopRecord
version	String	1.0
direction	String	Which media direction to stop record, in (from caller) or out (from agent)
sessionId	String	Session identifier.
requestId	String	Request identifier.

Example:

{
"method":"StopRecord",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d4b3da50-fdc6-11ea-a66b-ed36d1ad829b"
}

3.4.2. Response

Purpose: recording has stopped, AudioData events will no longer be sent

Property	Type	Value
method	String	Response
repsonseTo	String	StopRecord
version	String	1.0
sessionId	String	Session identifier.
requestId	String	Request identifier.
errors	Array	(optional) error event text
status	String	stopped

{"method":"Response",
"responseTo":"StopRecord",
"version":"1.0",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d4b3da50-fdc6-11ea-a66b-ed36d1ad829b",
"errors":[],
"status":"stopped"}

3.5. Play

3.5.1. Request

Purpose: schedule the attached audio for playback (audioData contains base64 encoded raw ulaw audio at 8khz). For streaming audio playback continue to schedule audio as each audio block is available. The audio is played from the queue in order of receipt.

Property	Type	Value
method	String	Play
version	String	1.0
direction	String	Which media direction to play, in (to the caller) or out (to the agent)
sessionId	String	Session identifier
requestId	String	Request identifier
audioData	String	Base64 encoded ulaw audio of arbitrary length
bargeIn	Boolean	enable bargeIn of play

{"method":"Play",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"audioData":"/w==",
"requestId":"d381c2f0-fdc6-11ea-a66b-ed36d1ad829b"}

3.5.2. Response

Purpose: notify the application that the play has been scheduled

Property	Type	Value
method	String	Response
responseTo	String	Play
version	String	1.0
sessionId	String	Session identifier

Property	Type	Value
requestId	String	Request identifier
errors	Array	(optional) error event text
status	String	playing

{"method":"Response",
"responseTo":"Play",
"version":"1.0",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d3821110-fdc6-11ea-a66b-ed36d1ad829b",
"errors":[],
"status":"playing"}

3.5.3. Event PlayComplete

Purpose: notify the application that all scheduled audios have completed playing

Property	Type	Value
method	String	Event
event	String	PlayComplete
version	String	1.0
direction	String	Which media direction was playing, in (to caller) or out (to agent)
sessionId	String	Session identifier
requestId	String	Request identifier

{"method":"Event",
"event":"PlayComplete",
"version":"1.0",

"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d3823820-fdc6-11ea-a66b-ed36d1ad829b"}

3.6. StopPlay

3.6.1. Request

Purpose: stop all scheduled audio plays immediately

Property	Type	Value
method	String	StopPlay
version	String	1.0
direction	String	Which media direction to stop playing, in (to caller) or out (to agent)
sessionId	String	Session identifier
requestId	String	Request identifier

Example:

{"method":"StopPlay",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d4b42870-fdc6-11ea-a66b-ed36d1ad829b"}

3.6.2. Response

Purpose: notify the application that all plays have stopped

Property	Type	Value
method	String	Response
responseTo	String	StopPlay
version	String	1.0

Property	Type	Value
sessionId	String	Session identifier
requestId	String	Request identifier
errors	Array	(optional) error event text
status	String	stopped

Example:
{"method":"Response",
"responseTo":"StopPlay",
"version":"1.0",
"sessionId":"838EEF3D-0C08-4B0F-85B2-D805A6B30927",
"requestId":"d4b42870-fdc6-11ea-a66b-ed36d1ad829b",
"errors":[],
"status":"stopped"}

3.7. ReceivedDTMF

3.7.1. Event

Purpose: Notify the connector of a received DTMF digit

Property	Type	Value
method	String	Event
event	String	ReceivedDTMF
direction	String	Channel DTMF received on “in” (from caller) or “out” (from agent)
value	String	Digit 0..9, * or #

4. References