IMAGE RECOGNITION SYSTEM USING IBM CLOUD VISUAL RECOGNITION.

Introduction :

To create a image recognition using IBM Cloud, including setting up an IBM Cloud account, creating a Visual Recognition service, creating IBM Cloud. In this integration, IBM Cloud Functions serves as the event-driven architecture where you can trigger image recognition functions in response to events such image uploads. When an event is triggered, it invokes a serverless function, which then utilizes Visual Recognition to analyze and interpret the images.

The application demonstrates an IBM Cloud Functions (based on Apache OpenWhisk) that gets an image from the Cloudant database and classifies it through Watson Visual Recognition. The use case demonstrates how actions work with data services and execute logic in response to Cloudant events.

One function, or action, is triggered by changes (in this use case, an upload of a document) in a Cloudant database. These documents are piped to another action that submits the image to Watson Visual recognition and upload a new document in Cloudant with the classifiers produced by Watson.

Steps :

1. Clone the repo

$ git clone https://github.com/IBM/serverless-image-recognition

2. Create IBM Cloud Services

Create a **Cloudant** instance and choose Use both legacy credentials and IAM for the *Available authentication method* option.

- Create credentials for this instance and copy the username and password in the local.env file in the value of CLOUDANT_USERNAME and CLOUDANT_PASSWORD.

- Launch the Cloudant web console and create a database named images and tags. Create Cloudant credentials using the IBM Cloud dashboard and place them in the local.env file.

Create a Watson Visual Recognition instance.

- Copy the API Key in the Credentials section and paste it in the local.env file in the value of WATSON_VISUAL_APIKEY

3. Deploy Cloud Functions

Create 2 databases in cloudant:

1. images
2. tags

Deploy through the IBM Cloud Functions console user interface

Choose "Start Creating" in the IBM Cloud Functions Dashboard. Then proceed to this deployment instructions using the UI.

You can also deploy them directly from the CLI by following the steps in the next section.

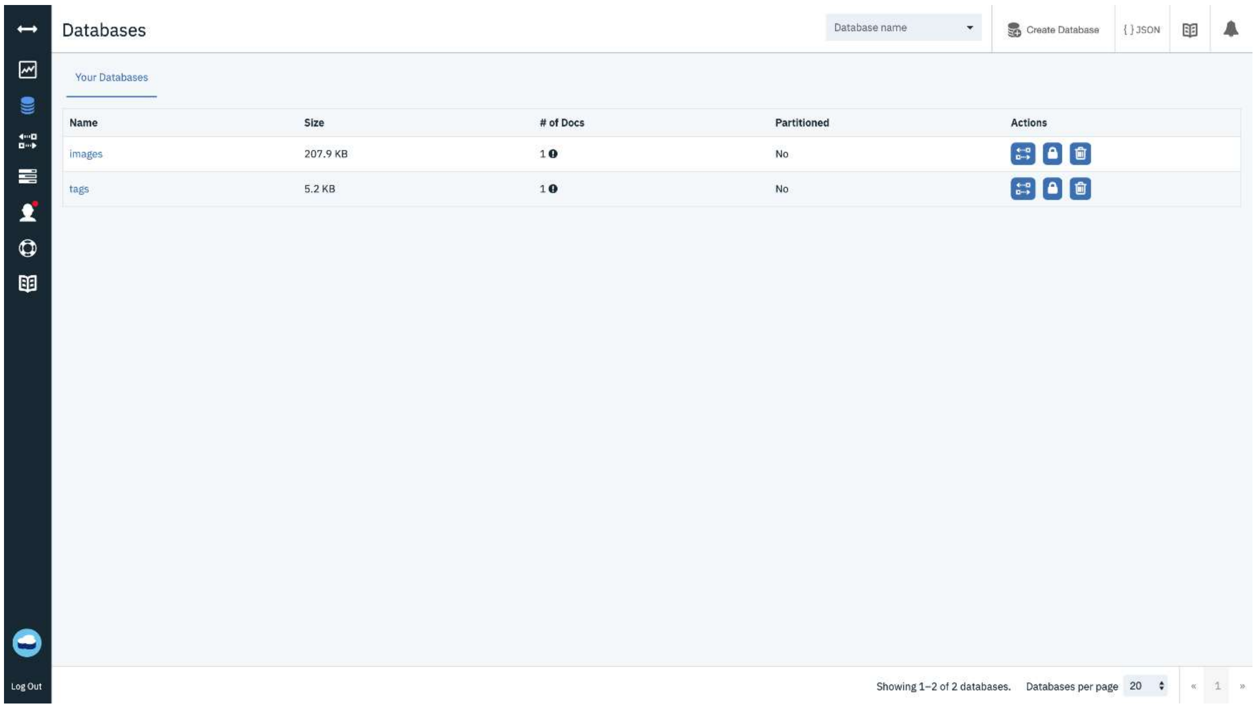
Deploy using the wskdeploy command line tool

This approach deploy the Cloud Functions with one command driven by the runtime-specific manifest file available in this repository.

Make sure you have the right environment variables in the local.env file. Export them in your terminal then deploy the Cloud Functions using wskdeploy. This uses the manifest.yaml file in this root directory.

```
$ source local.env

$ wskdeploy
```



Name	Size	# of Docs	Partitioned	Actions
images	207.9 KB	1	No	
tags	5.2 KB	1	No	

Showing 1–2 of 2 databases. Databases per page 20

4. Launch Application

Configure web/scripts/upload.js. Modify the lines for your Cloudant credentials.

```
let usernameCloudant = "YOUR_CLOUDANT_USERNAME"

let passwordCloudant = "YOUR_CLOUDANT_PASSWORD"
```

Run the Electron app or open the html file.

- Electron:

```
$ npm install
$ npm start
```

Alternative Deployment methods

Deploy manually with the ibmcloud wsk command line tool

This approach shows you how to deploy individual the packages, actions, triggers, and rules with CLI commands. It helps you understand and control the underlying deployment artifacts.

- Export credentials

```
$ source local.env
```

- Create Cloudant Binding

```
$ ibmcloud wsk package bind /whisk.system/cloudant serverless-pattern-cloudant-package \
-p username $CLOUDANT_USERNAME \
-p password $CLOUDANT_PASSWORD \
-p host ${CLOUDANT_USERNAME}.cloudant.com
```

- Create the Cloudant Trigger

  The trigger will listen to changes in the images database.

```
$ ibmcloud wsk trigger create update-trigger --feed serverless-pattern-cloudant-package/changes \
--param dbname images
```

- Create the Action

  The action executes your code. The code is already configured to use the Watson SDK.

```
$ ibmcloud wsk action create update-document-with-watson
actions/updateDocumentWithWatson.js \
--kind nodejs:8 \
--param USERNAME $CLOUDANT_USERNAME \
--param PASSWORD $CLOUDANT_PASSWORD \
--param DBNAME $CLOUDANT_IMAGE_DATABASE \
--param DBNAME_PROCESSED
$CLOUDANT_TAGS_DATABASE \
--param WATSON_VR_APIKEY $WATSON_VISUAL_APIKEY
```

- Create the Rule

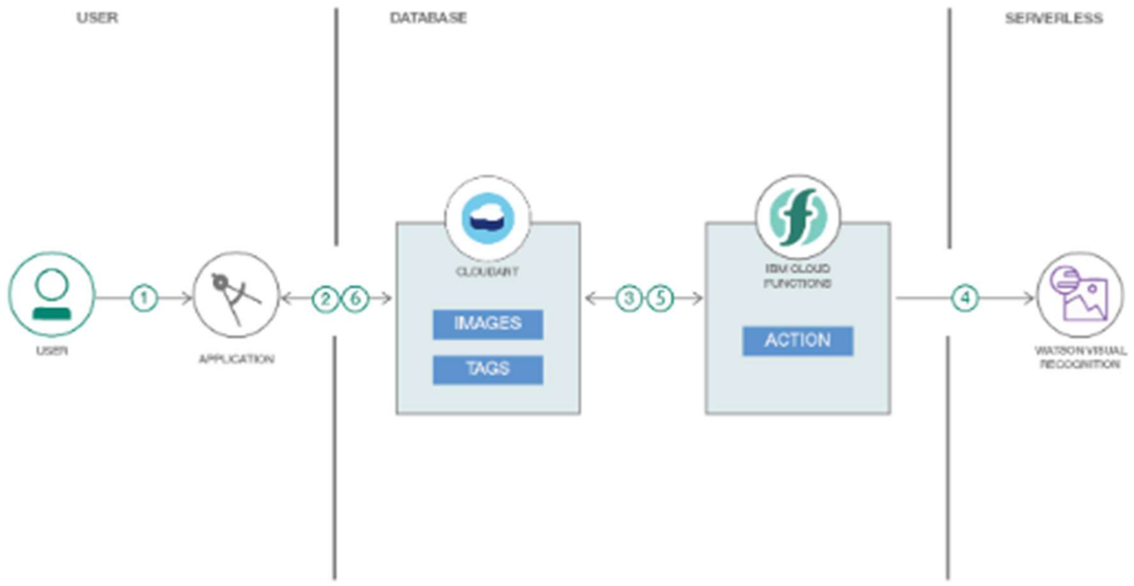
The rule will connect invoke the action when the trigger receives an event.

```
$ ibmcloud wsk rule create update-trigger-rule update-trigger update-
document-with-watson
```

- To delete them

```
$ ibmcloud wsk package delete serverless-pattern-cloudant-package
$ ibmcloud wsk trigger delete update-trigger
$ ibmcloud wsk action delete update-document-with-watson
$ ibmcloud wsk rule delete update-trigger-rule
```

Structure:



Program :

```
cache: pip

        before_install:
          - sudo apt-get install shellcheck
          - sudo pip install yamllint
          - git clone https://github.com/IBM/pattern-ci

        before_script:
          - "./pattern-ci/tests/shellcheck-lint.sh"
          - "./pattern-ci/tests/yaml-lint.sh"

        jobs:
          include:
   - script: echo "Lints passed."
```

gitignore :

```
        # Node.js

        # Logs
        logs
        *.log
        npm-debug.log*
        yarn-debug.log*
        yarn-error.log*

        # Runtime data
        pids
        *.pid
        *.seed
        *.pid.lock
```

```
# Directory for instrumented libs generated by jscoverage/JSCover
lib-cov

# Coverage directory used by tools like istanbul
coverage

# nyc test coverage
.nyc_output

# Grunt intermediate storage (http://gruntjs.com/creating-plugins#storing-
task-files)
.grunt

# Bower dependency directory (https://bower.io/)
bower_components

# node-waf configuration
.lock-wscript

# Compiled binary addons (https://nodejs.org/api/addons.html)
build/Release

# Dependency directories
node_modules/
jspm_packages/

# TypeScript v1 declaration files
typings/

# Optional npm cache directory
.npm

# Optional eslint cache
.eslintcache

# Optional REPL history
.node_repl_history
```

```
# Output of 'npm pack'
*.tgz

# Yarn Integrity file
.yarn-integrity

# dotenv environment variables file
.env

# parcel-bundler cache (https://parceljs.org/)
.cache

# next.js build output
.next

# nuxt.js build output
.nuxt

# vuepress build output
.vuepress/dist

# Serverless directories
.serverless


# Xcode
#
# gitignore contributors: remember to update Global/Xcode.gitignore,
Objective-C.gitignore & Swift.gitignore

## Build generated
build/
DerivedData/

## Various settings
*.pbxuser
```

```
!default.pbxuser
*.mode1v3
!default.mode1v3
*.mode2v3
!default.mode2v3
*.perspectivev3
!default.perspectivev3
xcuserdata/

## Other
*.moved-aside
*.xccheckout
*.xcscmblueprint

## Obj-C/Swift specific
*.hmap
*.ipa
*.dSYM.zip
*.dSYM

## Playgrounds
timeline.xctimeline
playground.xcworkspace

# Swift Package Manager
#
# Add this line if you want to avoid checking in source code from Swift
Package Manager dependencies.
# Packages/
# Package.pins
# Package.resolved
.build/

# CocoaPods
#
# We recommend against adding the Pods directory to your .gitignore.
However
```

```
        # you should judge for yourself, the pros and cons are mentioned at:
        # https://guides.cocoapods.org/using/using-cocoapods.html#should-i-check-
        the-pods-directory-into-source-control
        #
        # Pods/

        # Carthage
        #
        # Add this line if you want to avoid checking in source code from Carthage
        dependencies.
        # Carthage/Checkouts

        Carthage/Build

        # fastlane
        #
        # It is recommended to not store the screenshots in the git repo. Instead, use
        fastlane to re-generate the
        # screenshots whenever they are needed.
        # For more information about the recommended setup visit:
        # https://docs.fastlane.tools/best-practices/source-control/#source-control

        fastlane/report.xml
        fastlane/Preview.html
        fastlane/screenshots/**/*.png
fastlane/test_output
```

local :

```
        # Customize as needed
        export CLOUDANT_IMAGE_DATABASE=images
        export CLOUDANT_TAGS_DATABASE=tags
        export CLOUDANT_USERNAME=
        export CLOUDANT_PASSWORD=
export WATSON_VISUAL_APIKEY=
```

package :

```
{
  "name": "serverless-image-recognition",
  "version": "1.0.0",
  "description": "serverless",
  "main": "main.js",
  "directories": {
    "doc": "docs"
  },
  "scripts": {
    "start": "electron ."
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/AnthonyAmanse/serverless-image-
recognition.git"
  },
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/AnthonyAmanse/serverless-image-
recognition/issues"
  },
  "homepage": "https://github.com/AnthonyAmanse/serverless-image-
recognition#readme",
  "devDependencies": {
    "electron": "^7.1.2"
  }
}
```

manifest :

```
# Manifest for serverless-image-recognition
# Repo is located at https://github.com/IBM/serverless-image-recognition
# Installing openwhisk actions, triggers, and rules for Message Hub sample
app

# Deployment using this manifest file creates following OpenWhisk
components:
#   Package:    serverless-image-recognition
#   Action:     update-document-with-watson
#   Trigger:    update-trigger
#   Rule:       update-trigger-rule

# This manifest file expects following env. variables:
#   CLOUDANT_IMAGE_DATABASE
#   CLOUDANT_TAGS_DATABASE
#   CLOUDANT_USERNAME
#   CLOUDANT_PASSWORD
#   WATSON_VISUAL_APIKEY
---
packages:
  serverless-image-recognition:
    dependencies:
      # binding cloudant package named openwhisk-cloudant
      serverless-pattern-cloudant-package:
        location: /whisk.system/cloudant
        inputs:
          username: $CLOUDANT_USERNAME
          password: $CLOUDANT_PASSWORD
          host: ${CLOUDANT_USERNAME}.cloudant.com
    actions:
      # Action named "update-document-with-watson"
      # Creating action as a regular Node.js action
      update-document-with-watson:
        function: actions/updateDocumentWithWatson.js
        runtime: nodejs:8
        inputs:
          USERNAME: $CLOUDANT_USERNAME
          PASSWORD: $CLOUDANT_PASSWORD
          DBNAME: $CLOUDANT_IMAGE_DATABASE
          DBNAME_PROCESSED: $CLOUDANT_TAGS_DATABASE
          WATSON_VR_APIKEY: $WATSON_VISUAL_APIKEY
      triggers:
        # Creating the update-trigger trigger"
        update-trigger:
          feed: serverless-pattern-cloudant-package/changes
          inputs:
            dbname: $CLOUDANT_IMAGE_DATABASE
      rules:
        # Rule named "update-trigger-rule"
        # Creating the rule that links the trigger to the sequence
        update-trigger-rule:
          trigger: update-trigger
action: update-document-with-watson


  var Cloudant = require('@cloudant/cloudant');

  var fs = require('fs');

  var cloudant;

  var dbName;

  var dbNameProcessed;


  function main(params) {

      cloudant = Cloudant({account:params.USERNAME,
  password:params.PASSWORD});

      dbName = params.DBNAME;

      dbNameProcessed = params.DBNAME_PROCESSED;

      return new Promise(function(resolve, reject) {

          let mydb = cloudant.db.use(dbName);

          mydb.attachment.get(params.id, 'image', function(err, data) {
            if (err) {
                reject(err);
            } else {
                console.log(params)
                console.log(data)

resolve(processImageToWatson(data,params.id,params.WATSON_VR_API
KEY));
            }
        });
    });
}

function processImageToWatson(data,id,apikey) {
    let filename = __dirname + '/' + id;
    fs.writeFileSync(filename, data)
    let VisualRecognitionV3 = require('watson-developer-cloud/visual-
recognition/v3');
    var visualRecognition = new VisualRecognitionV3({
        version: '2018-03-19',
        iam_apikey: apikey,
    });

    var watsonvrparams = {
        images_file: fs.createReadStream(filename)
    };
    return new Promise(function(resolve, reject) {
        visualRecognition.classify(watsonvrparams, function(err, res) {
            if (err) {
                console.log(err);
                reject(err);
            } else {
                resolve(updateDocument(res,id));
            }
        });
    });
}

function updateDocument(watsonResult,id) {
    return new Promise(function(resolve, reject) {
        let mydb = cloudant.db.use(dbNameProcessed);
        var doc = {};
        doc._id = id
        doc.watsonResults = watsonResult.images[0].classifiers;
        mydb.insert(doc, function(err,body) {
            if (err) {
                reject(err);
            } else {
                console.log(body);
                resolve(body);
            }
            });
        });
    }
```

Conclusion :

Creating an image recognition system using the IBM Clous Visual Recognition in a cloud computing project allows for powerful image analysis capabilities that can be integrated into various applications and solutions.

Image recognition is a crucial application of artificial intelligence that involves the interpretation of images to extract meaningful information. IBM Cloud Visual Recognition is a state-of-the-art AI service that allows for image analysis, classification, and extraction of important features from images.

In conclusion, implementing an image recognition system using IBM Cloud Visual Recognition in a cloud computing project offers a powerful solution for automating image analysis, which can find applications in a wide range of domains, ultimately improving processes, decision-making and the user experiences.