LocalStack
Hands-On Tutorial

Credits to Harsh Mishra for sharing demo content

Basic Concepts

- **Host:** LocalStack can be accessed on `http(s)://localhost:4566` or `http(s)://localhost.localstack.cloud:4566`
- **Integration:** LocalStack provides a bunch of open-source integrations to connect with the host, such as `awslocal`, `tflocal`, `cdklocal`.
- **Configuration:** LocalStack is a highly configurable platform with **Network** configs, **Debug** settings, **Persistence** settings, and more!
- **IAM Enforcement**: By default, LocalStack is a “permit-all” system, i.e., no IAM restrictions. To strictly enforce IAM, you can set `ENFORCE_IAM=1`.
- **Event-Driven Architecture:** AWS's architecture is fundamentally event-driven, and you can use LocalStack logs to visualize them.

Connecting to AWS from Python

- AWS provides SDKs for various programming languages (Python, Java, .NET, Node.js, …)
- For Python, the official AWS SDK is called **`boto3`** → we will see this quite frequently in our sample apps
- Simple example of connecting to S3 - listing all buckets and objects:

```python
client = boto3.client("s3", endpoint_url="http://localhost:4566")

buckets = client.list_buckets()["Buckets"]
for bucket in buckets:
    print(f"Found bucket: {bucket['Name']}")
    objects = client.list_objects(Bucket=bucket['Name']).get("Contents", [])
    for obj in objects:
        print(f"Found object: s3://{bucket['Name']}/{obj['Key']}")
```

Run **`python list_s3_objects.py`** in the repo

Configuring the Dead Letter Config

- Dead Letter Configs (DLQ) handle events that cannot be processed successfully by a Lambda function.
- To setup the DLQ, navigate to `demo-2` and use the `run.sh` script to setup the SNS topic, SES identity & Lambda configuration..
- After setup, upload a non-image file to the S3 bucket and navigate to `http://localhost.localstack.cloud:4566/_aws/ses`.

```
$ awslocal sns create-topic --name failed-resize-topic
$ awslocal ses verify-email-identity --email my-email@example.com
$ awslocal sns subscribe …
$ awslocal lambda update-function-configuration …
```

Run **`demo-2/run.sh`** in the repo

Storing image metadata to DynamoDB table

- You can use a local DynamoDB table to store the image metadata. Navigate to demo-3 and copy the code to your original Lambda.

- To make sure that you can test your Lambdas continuously without having to re-start LocalStack or re-deploy Lambda, you can use **Hot Reloading**.

- However, let us first setup the DynamoDB table:

```
$ awslocal dynamodb create-table --table-name ImageMetaData …
$ awslocal dynamodb list-tables
$ awslocal dynamodb scan --table-name ImageMetaData
```

Run `demo-3/run.sh` in the repo

Hot Reloading the Lambda Functions

- What if we want to make changes to the existing functionality?
  - With **hot reloading**, changes are immediately reflected (→ fast feedback cycles!)
- We'll now extend the example and modify our Lambda function:
  - See the hint in the `demo-3/run.sh` script

```
$ awslocal lambda update-function-code --function-name … \
  --s3-bucket hot-reload --s3-key "$(pwd)/lambdas/list"
```

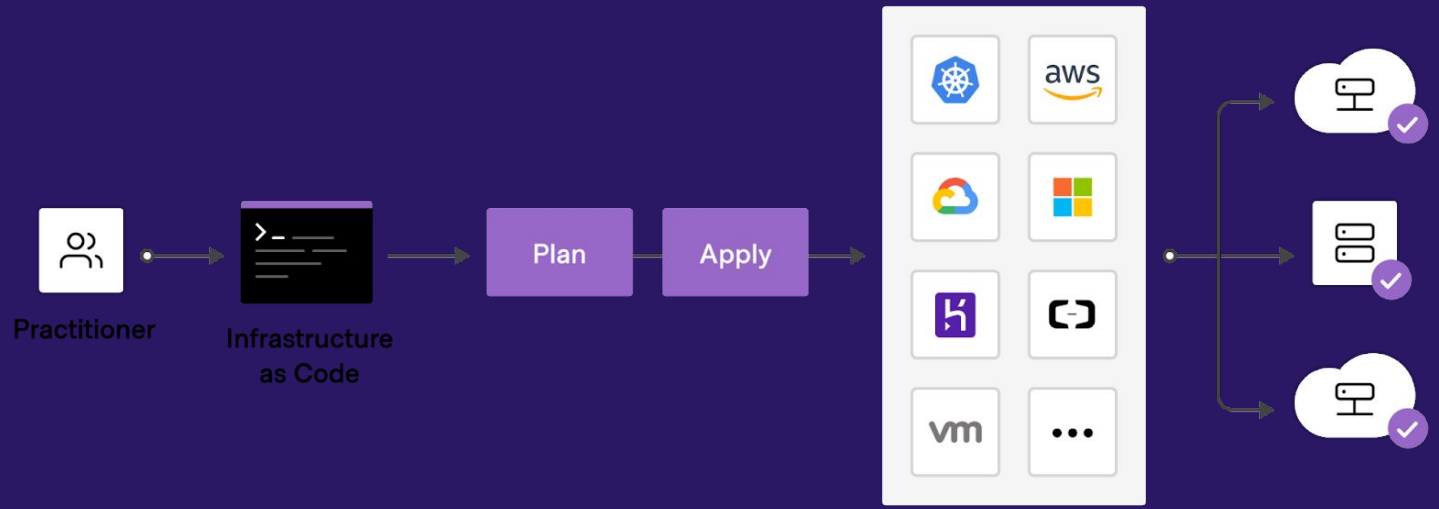
special bucket name

absolute path to Lambda code

- Modify the Lambda handler which will be reflected in the Lambda automatically — without updating your function configuration.

Introduction to Terraform

- Popular Infrastructure-as-Code (IaC) framework for deploying apps on cloud in an automated manner.
- Resources defined in a declarative way
    - Terraform creates a plan, which is then applied to create the resources



Deploying the application using Terraform

- `tflocal` script:
  - Install Terraform CLI
  - `pip install terraform-local`
  - Automatically configures the AWS provider to use the local endpoints under http://localhost:4566
  - Navigate to `demo-4` directory to start!
- Same workflow:
  - `tflocal init`
  - `tflocal plan`
  - `tflocal apply`

```
provider "aws" {
  access_key                  = "test"
  secret_key                  = "test"
  skip_credentials_validation = true
  skip_metadata_api_check     = true
  endpoints {
    acm = "http://localhost:4566"
    amplify = "http://localhost:4566"
    apigateway = "http://localhost:4566"
    apigatewayv2 = "http://localhost:4566"
    appconfig = "http://localhost:4566"
    ...
```

Run automated tests using `pytest`

- You can start a pytest workflow to run integration tests against locally deployed infrastructure. The pytest would:
  - Assert that the deployed resources are available.
  - Upload an image file and retrieve the resized image.
  - Assert that the resized image is smaller than the original image.

- You can run `pytest -v` in the root directory (after installing `pytest`) to run your tests:

```
gitpod /workspace/pycon-africa-24 (main) $ pytest --disable-warnings
================================== test session starts ==================================
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0
rootdir: /workspace/pycon-africa-24
plugins: typeguard-2.13.3, anyio-4.4.0, pylama-8.4.1
collected 1 item

tests/test_infra.py .                                                            [100%]

=============================== 1 passed, 6 warnings in 0.35s ===============================
```

Running LocalStack on GitHub Actions

- Install LocalStack on your GitHub Action workflow:

```
- name: Start LocalStack
  run: |
     pip install localstack awscli-local
     docker pull localstack/localstack
```

- Run LocalStack on your GitHub Action runner:

```
- name: Run LocalStack
  run: |
     localstack start -d
     localstack wait -t 15
```

Deploying your Terraform stack & running tests

- Deploy the Terraform stack using the `tflocal` script:

```
- name: Deploy the Terraform stack
  run: |
      cd demo-4
      tflocal init
      tflocal apply --auto-approve
```

- Run automated tests on the deployed infrastructure

```
- name: Run tests
  run: |
    pip install pytest
    pytest .
```