**Totesys Data Engineering Project**

**Infrastructure**

Here is a rough overview of the infrastructure that will need to be in place.

A diagram of a cloud computing system

Description automatically generated

The S3 buckets are used for data storage. In this case we will need three S3 buckets:

1. Ingestion Data
2. Processed Data
3. Terraform Data

The Lambda functions are used to implement code and perform actions on our data. The three lambda functions will be:

1. Ingestion Lambda
   1. Continually ingests all tables from the ‘totesys’ database.
   2. Data to be saved in files in the ingestion bucket.
   3. Data must be of a suitable format.
   4. Operates on a schedule; every 30 minutes.
   5. Logs progress to Cloudwatch.
   6. Triggers email alerts in the event of failures.
   7. Should follow security practices (prevents SQL injection and maintain password security).
2. Transformation Lambda
   1. Remodels data into a predefined schema suitable for a data warehouse and store the data in Parquet format.
   2. Stores data in the processed S3 bucket.
   3. Triggers automatically when it detects the completion of an ingested data job.
   4. Logs progress to Cloudwatch.
   5. Populates the dimension and fact tables of a single ‘star’ schema in the warehouse.
3. Loading Lambda
   1. Loads data into a prepared data warehouse at defined intervals.
   2. Monitored with Cloudwatch.

**Infrastructure as code**

Terraform is an open source tool. It is a declarative tool that allows us to specify the infrastructure we want as code. Terraform is a command line tool rather than a service – all the API calls to create resources will come from where we run it. Terraform is useful for creating resources on AWS – but can also support other cloud infrastructure providers too!

Terraform templates contain the resources that we want to create and can optionally contain variables that can be passed in when we run the tool.

Resources are declared with a ‘resource’ block that is followed by an identifier, enclosed in quotes (“”). Resources represent entities such as virtual machines, storage buckets, networks, DNS records and many others. Anything that can be created, updated, or deleted within a provider’s infrastructure can be represented as a resource in Terraform.

Here is an example of a resource:

resource "aws\_s3\_bucket" "my\_bucket" {

bucket = "my-unique-bucket-name"

acl = "private"

}

* **Aws\_s3\_bucket**: is the resource type – in this instance representing an AWS S3 bucket.
* **My\_bucket**: is the local name assigned to this specific instance of the resource block.
* **{}**: defines the rest of the code that we want to run; for example the NAME of the S3 bucket that will be hosted on AWS.

**Create Provider**

The provider configuration file ensures that Terraform knows which provider to use for the resources you will be managing in the infrastructure. Once the provider has been configured, you can proceed to create the resources such as the S3 buckets.

A screenshot of a bucket settings

Description automatically generatedIt would be prudent to have a pre-existing bucket for Terraform to store its tfstate file in. In this case there is the bucket; ‘terraform-xrs’.

Here is our current provider.tf file:

provider "aws" {

region = "eu-west-2"

}

terraform {

required\_providers {

aws = {

*source* = "hashicorp/aws"

*version* = "5.39.0"

}

}

backend "s3" {

bucket = "terraform-xrs"

key = "tf-state"

region = "eu-west-2"

}

}

The provider specifies that we will be using Amazon Web Services. The region will specify that London is the region resources are to be saved.

TFSTATE

It is essential to be able store backend information. The S3 backend stores state data in an S3 object set by the *key* parameter. In this case it has been hard coded to be ‘terraform-xrs’.

Here is the useful documentation on backends:

<https://developer.hashicorp.com/terraform/language/settings/backends/s3>

**Create Permissions**

We will now need to implement an IAM policy that will allow writing to and from the S3 bucket. Terraform, presently can not perform any actions as it does not have permission.

Because this is the providers.tf file it is unlikely to change throughout the project. Therefore it might be prudent to directly implement the IAM policy into the file. Here are some possible reasons for this:

1. Proximity to Provider Configuration: Keeping IAM policies within the same file as the provider configuration ensures that they are closely associated with the provider settings, making it easier to understand and manage the relationship between the provider and its associated permissions.
2. Simplicity: For simpler permissions that are specific tot eh provider configuration and are unlikely to be reused across multiple configurations, storing them directly in the ‘provider.tf’ file avoids unnecessary complexity introduced by separate policy files.
3. Ease of maintenance: If the permissions are closely tied to the AWS provider and are unlikely to change frequently, storing them in the provider.tf file can simplify maintenance by keeping all relevant configuration in one place.

resource "aws\_iam\_policy" "tfstate\_policy" {

name = "terraform-tfstate-policy"

description = "IAM policy for managing TFSTATE files"

policy = jsonencode({

*Version* = "2012-10-17"

*Statement* = [

{

*Effect* = "Allow"

*Action* = [

"s3:GetObject",

"s3:PutObject",

"s3:DeleteObject",

"s3:ListBucket"

]

*Resource* = "arn:aws:s3:::terraform-xrs/\*"

}

]

})

Here is a breakdown of the previous code.

* **Name:** specifies the name of the IAM policy, in this case it will be; ‘terraform-tfstate-policy’.
* **Description:** provides a description of what the IAM policy does.
* **Policy:** Defines the policy document in JSON format using the jsonencode() function. The policy allows the specified actions s3:getObject, s3:PutObject…etc….

This policy allows the IAM entity (user, group, or role) associated with this policy to perform the specified actions on objects within the ‘terraform-xrs’ bucket.

**Initialising Terraform**

* **‘terraform init’:** initialises terraform.
* **‘terraform plan’:** starts a terraform plan.
* **‘terraform apply’:** starts to apply terraform plan on Aws.

**Setting Up the S3 Buckets**

Our first task will be setting up the S3 buckets on AWS. The buckets will need a unique name and we will also need to be able to recall the names of the buckets after they have been created!

resource "aws\_s3\_bucket" "ingestion" {

bucket\_prefix = "ingestion-"

force\_destroy = true

}

Here the resource we are using is the aws\_s3\_bucket – which will allow us to create an S3 bucket. We then specify the name of the bucket. In this instance we use the bucket\_prefix which means the name of the bucket will have a unique set of numbers after ‘ingestion-……’.

Force\_Destroy is optional, however it allows us to delete all objects from the bucket when the bucket is destroyed so that it may be destroyed without error.

resource "aws\_s3\_bucket" "ingestion" {

bucket\_prefix = "ingestion-"

force\_destroy = true

}

# Creates Process Bucket:

resource "aws\_s3\_bucket" "process" {

bucket\_prefix = "process-"

force\_destroy = true

}

# Creates Storage Bucket:

resource "aws\_s3\_bucket" "storage" {

bucket\_prefix = "storage-"

force\_destroy = true

}

resource "aws\_ssm\_parameter" "ingestion\_bucket\_name" {

name = "/ingestion"

type = "String"

value = aws\_s3\_bucket.ingestion.bucket

}

resource "aws\_ssm\_parameter" "process\_bucket\_name" {

name = "/process"

type = "String"

value = aws\_s3\_bucket.process.bucket

}

resource "aws\_ssm\_parameter" "storage\_bucket\_name" {

name = "/storage"

type = "String"

value = aws\_s3\_bucket.storage.bucket

}

This code will setup three uniquely named S3 buckets. Additionally it will save the names of the buckets to AWS Parameter Store. This could be useful later!

**Implementing CI/CD**

Continuous integration is the process of merging changes into the main repository whilst using automated tests to verify it performs as expected. Changes should be small and integrated rather than having larger launch events. By introducing small changes we can lower the risk of creating code breaking bugs into the main codebase. Tests are usually run on a dedicated server so that the integration environment is common for every engineer working on the application.

Continuous delivery is a practice where the software can be deployed easily at any time with no manual intervention from engineers. As soon as the tests have passed after integrating a change then it should be possible to deploy that change. The idea is that the faster the change can be in front of the end users of the software then the faster feedback can be gathered and acted upon – one of the central ideas of Agile.

Continuous deployment is the idea that every change in version control is automatically deployed all the way through to production without any intervention from the engineers.

GitHub Actions

To get GitHub to trigger actions on a repository, the repository needs a directory named .github at the root level, and inside that, a workflows directory. Workflow is the name given to what might be called a pipeline with other providers. There may be multiple pipelines for a project. The repo may be responsible for multiple applications deployed separately, for example, or one could separate the CI/CD parts of the pipeline. Each workflow needs a YAML file to configure.

To begin we should name our file ‘test-and-deploy.yml’. This must be installed in the root of the directory. A black background with red text

Description automatically generated

The pipeline can be triggered by various events in the GitHub architecture. Commonly for CI/CD pipelines, you would use push and pull\_requests to initiate the pipeline.

name: Test & Deploy

on: [push]

There are some checks that one may wish to run every time the code is pushed into the main repository. This is part of the continuous integration of the code into the repository. These may include unit tests, or using tools for static code analysis – tools that check code without running it – such as linting, assessing vulnerabilities and other code quality checks.

Other processes may only make sense to run once we are happy that the code has passed previous checks, plus any other manual code reviews that need to be undertaken anyway – this might involve a staging area so checks for things like UX and accessibility can be signed off, or it might mean straight into production as we will do – that’s continuous deployment.

As the workflow will handle all of the CI/CD, the workflow file can be configured to run whenever a specific event happens. In this case, some code being pushed to the main branch. This will allow us to work on other branches without the pipeline being initiated.

on:

push:

branches:

- main

Jobs

GitHub Actions allows you to define separate tasks, or jobs that you may wish to undertake in the pipeline. With this structure we can separate the processes from each other, and provide different contexts or environments for them to be undertaken in.

jobs:

run-tests:

name: run-tests

runs-on: ubuntu-latest

steps:

- name: Checkout Repo

uses: actions/checkout@v4

On this run-tests job property we have defined a human-readable name, used by the GitHub UI to inform us about the pipeline’s progress. Additionally an OS has been provided for the job to be run on. Typically we should be emulating the environment of our deployment. On GitHub Actions, Linux machines are the cheapest, so ubuntu-latest would be a sensible option.

For our project, initially we want to ‘just’ setup a basic terraform deployment – we can run checks later.

jobs:

deploy:

runs-on: ubuntu-latest

steps:

**Note:** There are common github actions provided by the community:

<https://github.com/actions/setup-python>

This is the ‘setup-python’ github action that we can use for this project.

Jobs can be further broken down into steps that can be undertaken sequentially. The first step of our code is specific to GitHub Actions – it is necessary to **checkout** the code that we are working with. In a GitHub CI/CD workflow, checkout out the code is necessary because the CI/CD pipeline needs access to the source code of the repository in order to perform various tasks such as running tests. The source code of a project refers to the human-readable files and documents that contain the instructions and logic necessary to build and run the software application.

Next we will need to setup a python interpreter. There is a common GitHub-made Action we can use. We can provide information to the action using the ‘with’ key. Here we can use the latest Python version (at time of writing).

- name: Use Python

uses: actions/setup-python@v5

with:

python-version: '3.12'

Make

The next task our workflow needs is to create a virtual environment, install the project requirements and run the tests with the correct PYTHONPATH. It is possible to simplify instructions using a Makefile. We can use makefiles to run a sequence of commands.

The Makefile is a repository for code at the root of the project that is accessed by the make command. Make is installed on MacOS along with the XCode utility.

- name: Application Requirements

run: make requirements

This command will run our makefile.

Deployment

Assuming our continuous integration checks work, we can rigger a deployment action. We can create a job which runs a deployment script file. The one thing we need to do differently is how we authenticate to AWS – the virtual machine will not have access to our AWS credential files.

In order to deploy, we will need to provide the deployment script with access to the AWS Access Key ID and Secret Key.

**Note:** we do not want these to be hard coded into our files! This would create a security vulnerability. The solution is to declare ‘Secrets’ from within the Github console. A better way is to make these specific.

We can use this github actions: https://github.com/aws-actions/configure-aws-credentials

In order to deploy to AWS we need the following secrets:

* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY
* AWS\_REGION

- name: Implement AWS Credentials

uses: aws-actions/configure-aws-credentials@v4

with:

aws-region: ${{ secrets.AWS\_REGION }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

We now need to add our secrets to github. To do this, simply go to the repository you are working in and add the secrets:

A screenshot of a computer

Description automatically generated

Here is the final CI/CD file:

name: Test & Deploy

# Specify deployment intialisation.

on:

push:

branches:

- main

jobs:

# Tests

# Deploy

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout Repo

uses: actions/checkout@v4

- name: Use Python

uses: actions/setup-python@v5

with:

python-version: '3.12'

- name: Application Requirements

run: make requirements

- name: Implement AWS Credentials

uses: aws-actions/configure-aws-credentials@v4

with:

aws-region: ${{ secrets.AWS\_REGION }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

- name: Deploy Terraform

uses: hashicorp/setup-terraform@v3

- name: Terraform Init

working-directory: terraform

run: terraform init

- name: Terraform Plan

working-directory: terraform

run: terraform plan

- name: Terraform Apply

working-directory: terraform

run: terraform apply -auto-approve

run: terraform init

- name: Terraform Plan

working-directory: terraform

run: terraform plan

- name: Terraform apply

working-directory: terraform

run: terraform apply -auto-approve

This ultimately creates a very basic setup. One of the advantages of this is that the tfstate is stored in a predefined file which can be read. The advantage of this, is that every time the CI/CD pipeline is initiated, terraform will read from the tfstate to see which jobs need to be completed. This means that if we already have said up an ‘ingestion’ bucket; it will not be setup again.

We now have a working CI/CD pipeline:

A screenshot of a computer

Description automatically generated

We can verify that three buckets have been created; ingestion, process and storage.

A screenshot of a computer

Description automatically generated

Now that we have a working CI/CD pipeline with a very basic terraform initialisation, we can begin to think about building our first Lambda and incorporating it into the CI/CD pipeline.

**Lambda 1 – Ingestion**

We know that the first lambda should;

1. Continually ingests all tables from the ‘totesys’ database.
2. Data to be saved in files in the ingestion bucket.
3. Data must be of a suitable format.
4. Operates on a schedule; every 30 minutes.
5. Logs progress to Cloudwatch.
6. Triggers email alerts in the event of failures.
7. Should follow security practices (prevents SQL injection and maintain password security).

**Creating Functions**

To implement the first lambda successfully, it would be useful to setup some helper functions. For basic functionality we need to consider:

1. A function that can access the time stored in AWS Parameter Store.
2. A function that could update the time stored in AWS Parameter Store.
3. A function that can scan the totesys database and extract information updated since last update.
4. A function that can take our data and store it in an S3 bucket.

Following the creation of the functions, they will need to be tested. All functions should be developed through Test Driven Development to ensure they function correctly.

**Function 1 – “**get\_bucket\_names()”

One of the very first functions we should consider, is a function that is able to read the tfstate file and is able to extract the names of the buckets that have been created.

Fortunately during CI/CD deployment, Terraform wrote the names of the buckets to an AWS Parameter Store file. Therefore it should be a simple matter of retrieving them.

*def* get\_bucket\_names():

"""

Args:

------

None.

Returns:

------

Object containing names of S3 buckets. Example:

{'ingestion': 'ingestion-20240304201826545600000001',

'process': 'process-20240304201826547200000003',

'storage': 'storage-20240304201826546800000002'}

"""

client = boto3.client('ssm')

bucket\_obj = {'ingestion': None, 'process': None,

'storage': None}

for name in bucket\_obj:

bucket\_obj[name] = client.get\_parameter(

*Name*=*f*"/{name}")['Parameter']['Value']

return bucket\_obj

**Function 2 –** ‘get\_aws\_time()’

In order to determine which data we need to lift out of the totesys database, we will need to know the time we last scanned the database. To do this we will need to extract the time out of the AWS Parameter Store.

*def* get\_aws\_time():

"""

## Args:

None.

---

## Returns:

---

- datetime object stored in aws parameters store.

Returns a message: "File path created: {date\_str}. File added!"

This function accesses the time stored in AWS Parameter Store.

This function will return a datetime OBJECT.

"""

client = boto3.client('ssm')

str = client.get\_parameter(*Name*='/time')['Parameter']['Value']

return datetime.strptime(str, '%Y-%m-%d %H:%M:%S.%f')

**Function 3 –** ‘update\_aws\_time()’

Assuming we can extract the time. Note – the default time is set during the Terraform Init phase (to around 1900). Setting the default time when Terraform starts should ensure that ALL data is pulled from the database the first time our Lambda function will run.

After the Lambda function runs, we then need to update the time in the AWS Parameter Store. To do this we need a new function.