Sebastian Kosteczka DevOps test

1. Preparations

Solutions are written in Python, based on some pip packages and MongoDB. To run all scripts you will need:

- Python 3.10
- Pip with pip install pymongo Faker PyYAML Flask playwright
- MongoDB 6.x.x
- Playwright

I used virtualenv to separate this project from my typical workspace, but this is not required.

To enable MongoDB auth, you need to edit /etc/mongod.conf and add in the security section:

security:

authorization: enabled

2. Run scripts

a) MongoDB needs to be installed and running. You don't need to create anything manually with mongosh, but in some distros, you need to have mongod in the background (if you don't have a service for it).
b) Run "python create-user-db.py" – it will create the database and user
c) Run "python populate-data.py 20000" – it will create randomize entries in the collection
d) Run "python restapi.py" to start the Flask handler in the dev mode and from now on you can GET your endpoints.

We have these endpoints available:

http://localhost:5000/transactions/90

http://localhost:5000/transactions/90/card/Visa

http://localhost:5000/transactions/90/country/US

http://localhost:5000/transactions/90/amount/100.0/999.0

You can use curl or your browser.

3. Test REST API

a) Run “python test_playwright.py” to start a super simple test for each endpoint. It can also detect if the REST API is down.

Of course, if you would like to, you can change details about the MongoDB connection in the db_envs.yaml file.

4. Notes

a) I used Faker to generate random data because is easy to use and fast and covers all my needs.
b) The playwright test is super simple, but to be honest, I never write them myself. Most of the time testers or developers were responsible for it, so I needed sometimes only debug them or update or fix some small issues. And probably they were never written in Python (most of the time TS).
c) As I mentioned at 1st stage, I never was super into Python, so the code can be a little dirty, for certainly contains a DRY code.
d) I left Flask in the dev mode to be sure that it can be easily started for test purposes.
e) I didn’t merge scripts into one with a fancy menu to select commands, but that’s something was would be nice to add.
f) I resued “_id” in the MongoDB collection to avoid duplicates from “id” or need to remove it from the generator.

5. AWS deployment

I have here two scenarios which I would consider as a good approach to this task:

a) Lambda
b) EKS

What can be common for both solutions:

- VPC
- MongoDB Atlas
- Terraform to deploy infrastructure for the solution

- CI/CD – GitHub Actions, Jenkins, GitLab or even CodeCommit, AWS CodeBuild, AWS CodeDeploy, and AWS CodePipeline
- HashiCorp Vault or AWS Secrets Manager for a more secure store for sensitive data
- AWS Lambda doesn't natively support running Python apps like Flask. But can run Docker. We need also a Docker image for EKS, so we can create a Dockerfile for both solutions.
- for prod purposes we should switch to the WSGI server, in this case, Gunicorn can be the best approach
- container registry

Components – EKS

a) IAM policies
b) VPC with public and private subnets, NAT gateway
c) ECR
d) MongoDB Atlas
e) EKS node solution or with Fargate profiles
f) K8s addons via EKS addons or terraform helm provider – e.g. ingress controller
g) (optional) Route53 or any other DNS

Components – Lambda

a) IAM policies
b) VPC with public and private subnets, NAT gateway
c) ECR
d) MongoDB Atlas
e) Lambda function
f) API Gateway
g) (optional) Route53 or any other DNS

Steps

1. We need to write a Terraform plan for infrastructure based on Lambda or EKS. In both cases, we can use modules to create specific parts (e.g. VPC, EKS, Lambda, API Gateway) or just separate .tf files for each AWS resource. To be sure that we can handle and deploy all stages, we should have separate groups of variables to based on them create separate envs. To manage this even better, we can use a solution like Terragrunt. Then we can have providers and modules common for our envs, but specific envs stored in e.g. production/terragrunt.hcl, qa/terragrunt.hcl.
2. We should add our terraform init, plan and apply to our CI. Fail on any errors from init or plan. We should also include tfsec, tflint and checkov.
3. Infrastructure deployment should be based on branches or/and tags.
4. We need to prepare a Dockerfile file with our code, bundled with Gunicorn.

5. We need to build and push this image to ECR via our selected CI solution.

Deploy steps – EKS

1. In this case I would like to go for Helm charts and deploy them via CI/CD to our EKS cluster. Because we don't need to have any PV/PVC, we can use HPA to scale our deployment/sts.
2. For this we should use helm cli in our CI/CD or we can use helm provider in the Terraform plan, but I wouldn't mix app deployments with infrastructure deployments.
3. We can separate envs by unique namespaces (on normal nodes or Fargate profiles) or we can have separate clusters for each env. Even with security based on namespaces and network policies, we still should have considered separating pre-prod and prod from dev and QA by separate clusters.
4. If we create separate clusters, we should also have at least separate subnets for each or in a better-case scenario, individual VPCs.
5. We can use Nginx ingress or Traefik, but I would stick to using Istio.

Deploy steps – Lambda

1. After the infrastructure apply is completed, and we have our docker image tagged and pushed to ECR, we should deploy it to our Lambda function. In this case, we don't need to worry about scaling because we are using the serverless solution.
2. Of course here we can use Terraform once again, but we can also stick to CI/CD based on aws-cli and aws lambda update-function-code.
3. Each env should have its own lambda function and api gateway.

Summary

1. We should stick to minimising exposing our solution to the Internet – with Lambda only API Gateway endpoints should be accessible, with EKS only ALB URLs. We should not only manage subnets properly but also security groups.
2. We should stick to minimizing IAM policy permissions to only needed.
3. If we can, we should store sensitive data (e.g. MongoDB user/pass) in Vault or AWS SM.
4. If we'll stick to one cluster for more than 1 env, we should always use separate namespaces, RBAC resources and policies (e.g. network).
5. Playwright and Terraform tests should be a part of our CI/CD each time, even on dev env.
6. We should also create monitoring for this solution – to follow errors and performance insights.
7. We also should minimise all manual actions or custom resources (like custom AMIs). We'll avoid a lot of inconsistent errors.