Woven Interview

Engineer Challenge. Point 1.

Li Lin, Pau
lilin.pau@gmail.com

November 2022

1 Problem Description

You are building a model inference application and a REST API application in which the access endpoint and documentation will be provided to external clients so that they can upload an input image and get predicted results via the CURL command. You are decided to work based on software development best practices that balance implementation and schedule.

1. Create a diagram that shows the platform design consists of Kubernetes components, request/response flow, connection protocol, etc. Please draw components as many as you consider necessary to share with the team and reflect your knowledge level.

2. If a Cloud machine instance, such as EC2, is preferable for your design, please provide support reason why we should select that instance type.

3. Team members are not familiar with Kubernetes, Cloud, and networks. A good detailed explanation of the diagram would help to educate team members.

4. Points to consider while designing the platform and creating the diagram:

   (a) Optimizing machines' resources based on GPU and CPU needed.

   (b) Scalability of deployed applications to support a high volume of requests. Dynamically scale is preferable.

2 Architecture overview

Given the problem statement, we know that the objective of this project is to provide an architecture suitable to deploy and serve inferences in a production level. This means the solution should range from the development of new models, to monitoring the health of the project, to serving the inferences, down to storing results and analyzing the effectiveness of our solution. To provide and overview of an initial draft of the solution, we are going to make use the Figure 1 to explain the different components the full solution would look like. **To have a closer look, this figure can be found in pictures/initial_architecture_solution.png** The next subsections will describe the different components of this draft.

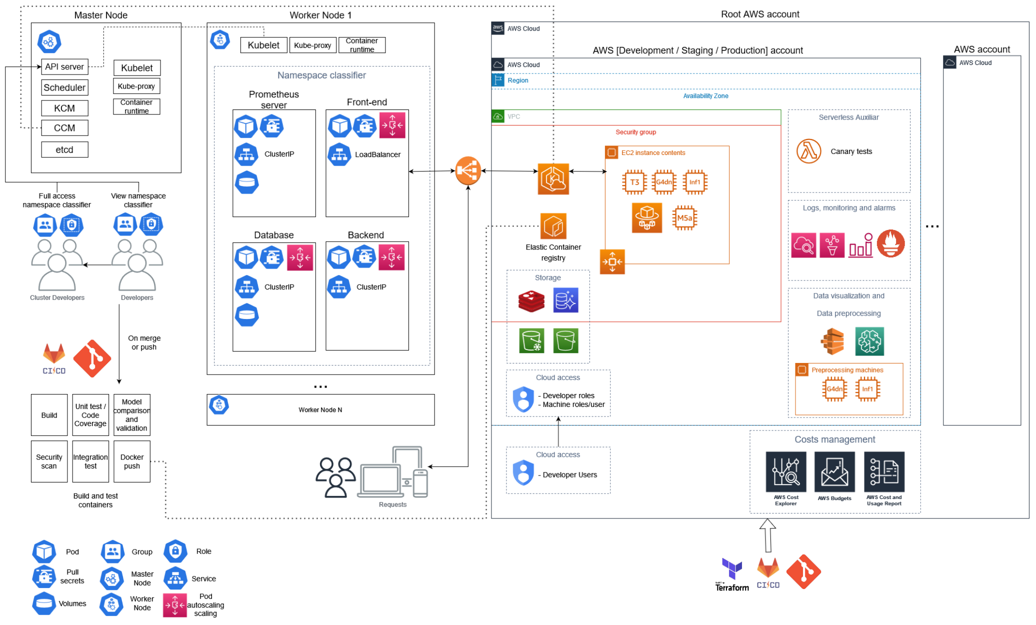

Figure 1: Initial K8S and AWS architecture drafts .

2.1 AWS account structure, developer access and deployment of the infrastructure

Suppose that we want to deploy this project in the cloud - specifically on AWS - we would like to have an account structure in which we will host different environments of the project. Specifically, we would like to have a umbrella account (root account) in which other *"sub-accounts"* will be created under it. These *"sub-accounts"* will represent different environments. These environments are: development (where engineers will develop and test new features), staging (where the candidate version is tested and delivered to potential third party developer groups) and production (where the project is served to the client).

Since we want to test our new versions in environments very similar to production, all environments are very similar to each other. Moreover, given that the infrastructure can also be hard to maintain and version, infrastructure as code will be required to launch these environments. For this task I would use *Terraform* to maintain the infrastructure since it is a well established framework and it has more support and documentation than other frameworks. I would have a separate git repository with the declarations of the infrastructure, and run it through CI/CD whenever a new change is required (probably called through merge acceptance of new versions).

I envision that all the developer users will be created on the main account with very limited permissions on the root account. These users would be able to take *developer roles* on different environments, depending on the responsibility of the user.

It is critical to have the root account credentials well secured and use M2F in all accounts.

There is also debate to have all configs files and data on the main account, but I am not sure how I feel about that yet.

2.2 Kubernetes Management

Different groups, users and roles must be in place in order to keep the kubernetes cluster secure. I would suggest to have the project inside a namespace, and give engineers with more proficiency on kubernetes full access to this namespace, the other engineers would have read permissions, to monitor the state of the deployments. Deployments and services would be performed through the CI/CD, which will be another separate user that can create and destroy deployments.

2.3 AWS EKS structure and Computing options

Although to me the configuration of the cluster is still not 100% clear, I know that we would like to run one cluster per environment. The EKS cluster will be configured by the full access engineers, and the deployments and services would be changes through the CI/CD pipeline. The kinds of pods that we may have inside the namespace is discussed in Section 3.

We would use manage node group to manage our EC2 computing instances. As for computing EC2 machines we have multiple options. The option choosen may depend on the amount of traffic that we expect to receive, the relevance of real-time and the pricing.

If we expect to receive very low traffic, and real-time is not a concern, we could have the model on *fargate*. *Fargate* is a serverless solution in which allow us to run code (in our case inferences) and pay only for the amount of resources requested. This would be the cheapest solution by a large margin. The scaling of *fargate* is also quite straight forward which we could serve to new users, as long as we handle our builds properly to not have cold starts. *Lambda* could also be a solution, however we would have to fit the entire project within 250MB and we may experience cold starts. I would only recommend *Lambda* it if not having real-time is a problem and we do not expect to scale up (since it is the cheapest way of deploying it), but overall seems like a poor solution.

We could use CPU machines in case that we see that the load that we receive is not as high, which would allow us to not use GPU machines, which are fairly expensive compared to CPU machines.

As personal recommendation, if our model is a single model without complex or abnormal structures, I would recommend to use Inferentia machines. Inferentia machines, AWS Inferentia is a machine learning inference chip, custom designed by AWS to deliver high throughput, low latency inference performance at an extremely low cost. AWS Inferentia will support the TensorFlow, Apache MXNet, and PyTorch deep learning frameworks, as well as models that use the ONNX format.[1]. It reduces the cost per image and sometimes it speeds up the process. The main drawback is that it requires to convert the models using the SDK neuron [2].

Finally, I would suggest using gpu g4dn.xlarge. Since what we are interested on fit as many models into the GPU VRAM, it makes sense to have multiple g4dn.xlarge and scale horizontally rather than the same one with more CPU. Notice that this machine is much expensier than the inferentia machines (g4dn.xlarge 0.71 cents, while inf1.xlarge 0.31 cents [3]. This solution is only good if we do not care about costs and we do not have time to make the sdk neuron conversions.

Regarding on scaling of machines, it may be handled by autoscaling groups created when we create of EKS cluster. Regarding scaling up or down, Dev and stag environment may not more than 2 machines as maxSize. The amount of machines that we can scale horizontally in the production environment depends on the budget of our project.

Other services besides the model such as front-end or database can run on T3 or M5 machines, which are all purpose machines.

2.4 Code pipelines and submission

The submissions of code would be performed on git repositories such *Gitlab*. New features would be developed on parallel branches, and whenever a new merge request happens, the different tests of the project would run over a build of the branch automatically. If the unit-test and end-to-end passes successfully, a peer review would be required. If the peer review is accepted and there are no merge conflicts, it would build the project, push it to the *ECR* and test it once more on the development environment. If all test seems to be fine and the deployment is healthy, it would automatically be build and tested on the staging environment, if everything seems fine, it would slowly replace the old builds for the new build on staging.

Deploying to production can be automatic as staging or it may require human intervention. It depends on two factors:

- The confidence that the team has on the test cases.
- The maturity of the project.

I would suggest to not deploy to production automatically since the project seems to be just starting.

2.5 Storage options

To store the data obtained from the user, it would be advisable to stay within the AWS ecosystem. S3 and S3 glaciers are great options to store images since they are easily configurable and accessible from other services such as amazon sagemaker. Database wise, I would prefer to use dynamoDB or redis, depending on the requirements of the project. While dynamoDB would fit greatly, redis would provide us with speed in case we need it. Slightly more explanation is provided on section 3.3. These databases may run on a pod from our EKS cluster, using stable CPU machines such as M5.

2.6 Logs, monitoring and alarms

For logging, monitoring and alarms we can use the native solutions from AWS. Cloudwatch would provide us with basic information of our infrastructure such as petitions requested, CPU usage and potential failures. Alongside, we could use *Prometheus* to get more information from inside the pods. Alarms may rise when something critical happens or when a system has an unexpected error.

2.7 Data exploration and preprocessing

Once our model is out on production serving classifications. We could analyze the results using *amazon sagemaker notebooks*, which has a great integration reading data from AWS databases and s3. We can also execute prepossessing procedures or redo the classification in batch to save computational costs. These executions can be handled using *batch jobs* in a different cluster where the machines only turn on when it is required. We could also use on-spot machines to run our process whenever there are machines available. Since this may not be a critical execution, it may be slower but it may reduce costs.

2.8 Auxiliar functions

There may be auxiliar functions to help ensure the health of our project. Right now the only one I can think of is having a canary test which executes our servers once in a periodic amount of time. If the results are not consistent it may rise a a flag notifying the engineers.

More auxiliar functions may appear during the development of the project.

3 Potential services within the pods

Since the requirements of the project are a little bit ambiguous and open ended, it is hard to determine specifically which requirements the potential client would like to set for this deployment. However, we can hypothesize that some core functionalities may be included in the project.

3.1 Front-end pod

The front-end pod may be the interface in which the consumer ask for inferences. The front-end may run over a a fast, well established web framework (such as fastAPI, Djando or flask). We could expect the front-end to have some sort of authentication requests, prediction requests, queueing and consult previous results using HTTP get and post requests.

Another option would be to use services that AWS provides, such as *cognito* to perform the authentication and *SQS* to queue the petitions from the client. Drawbacks of this approach may be that it may be hard to monitor besides the logs provided on *cloudwatch* and we would be providing limited AWS access to our clients, which if it is not implemented correctly it may become a security pitfall.

3.2 Back-end pod

The back-end pod may be the one producing the inference and the logic of the application. In this case, we can also use a web-framework to serve the back-end, however there are other methods specific for deploying ML models that we could use, such as KubeFlow or Sagemaker notebooks. TorchServe is also an interesting option, but it feels to be less common than the other alternatives [just a feeling].

Whichever framework it is decided to work with, we could expect that the back-end is hidden from the outside world (maybe using clusterIP services and cross communication of pods through kube-proxy). This back-end may not have access to Internet. We should also expect from it to be able perform inferences over the desired images of the users and return the results in the desired format for the user. We can also expect to be able request to store data in a database and images on s3. If the client charges for requests here is would be a good point to manage that. The images stored on s3 would allow us to analyze the results, see if our model is drifting results overtime and gather additional training data.

3.3 Database and Prometheus

As the database concerns, there are multiple available choices but basically it boils down to a flavour of non-relational database or a relational database. Non-relational databases would allow us to scale horizontally much easier and have a more flexible structure on our database. If we know the potential characteristics of our data and define a reasonable structure, relational databases may be easier to handle and maintain. Overall I would be inclined towards a non-relational database such as Redis or DynamoDB, given that they can handle and store huge amounts of data and the focus on analyzing or reading the data is more a secondary process.

Prometheus is a stable logging solution which allow us to have more information about our infrastructure. It allow us to store metrics from our system, for the web server it might be request times, for the database it might be number of active connections or number of active queries and for our back-end the amount of calls, gpu usage, errors or execution times. In my opinion it would be wise to set it up through the system to be able to monitor and state the health of the operations.

Both cases may use clusterIP to allow other services from the cluster to talk to them. These services may have restricted internet access to communicate with AWS. These pods may also have volumes to store temporally data.

References

[1] https://aws.amazon.com/about-aws/whats-new/2018/11/announcing-amazon-inferentia-machine-learning-inference-microchip/

[2] https://aws.amazon.com/machine-learning/neuron/

[3] https://aws.amazon.com/ec2/pricing/on-demand/