AWS Developer Tools Blog

AWS Batch Application Orchestration using AWS Fargate

by Siva Ramani | on 11 OCT 2021 | in Advanced (300), Amazon DynamoDB, Amazon EC2, Amazon EC2 Container Service, Amazon VPC, Architecture, AWS Batch, AWS Fargate, AWS SDK for Java, Containers, Serverless, Technical How-to | Permalink | Comments | Share

Many customers prefer to use Docker images with AWS Batch and AWS Cloudformation for cost-effective and faster processing of complex jobs. To run batch workloads in the cloud, customers have to consider various orchestration needs, such as queueing workloads, submitting to a compute resource, prioritizing jobs, handling dependencies and retries, scaling compute, and tracking utilization and resource management. While AWS Batch simplifies all the queuing, scheduling, and lifecycle management for customers, and even provisions and manages compute in the customer account, customers continue to look for even more time-efficient and simpler workflows to get their application jobs up and running in minutes.

In a previous version of this blog, we showed how to spin up the AWS Batch infrastructure with Managed EC2 compute environment. With fully serverless batch computing with AWS Batch Support for AWS Fargate introduced last year, AWS Fargate can be used with AWS Batch to run containers without having to manage servers or clusters of Amazon EC2 instances. This post provides a file processing implementation using Docker images and Amazon S3, AWS Lambda, Amazon DynamoDB, and AWS Batch. In the example used, the user uploads a CSV file into an Amazon S3 bucket, which is processed by AWS Batch as a job. These jobs can be packaged as Docker containers and executed on Amazon EC2 and Amazon ECS.

The following steps provide an overview of this implementation:

1. AWS CloudFormation template launches the S3 bucket that stores the CSV files along with other necessary infrastructure.
2. The Amazon S3 file event notification executes an AWS Lambda function that starts an AWS Batch job.
3. AWS Batch executes the job as a Docker container.
4. A Python-based program reads the contents in the Amazon S3 bucket, parses each row, and updates an Amazon DynamoDB table.
5. DynamoDB stores each processed row from the CSV.

Prerequisites

- Make sure to have Docker installed and running on your machine. Use Docker Desktop and Desktop Enterprise to install and configure Docker.
- Set up your AWS CLI. For steps, see Getting Started (AWS CLI).

Walkthrough

Below steps explains how to download, build the code and deploying the infrastructure.

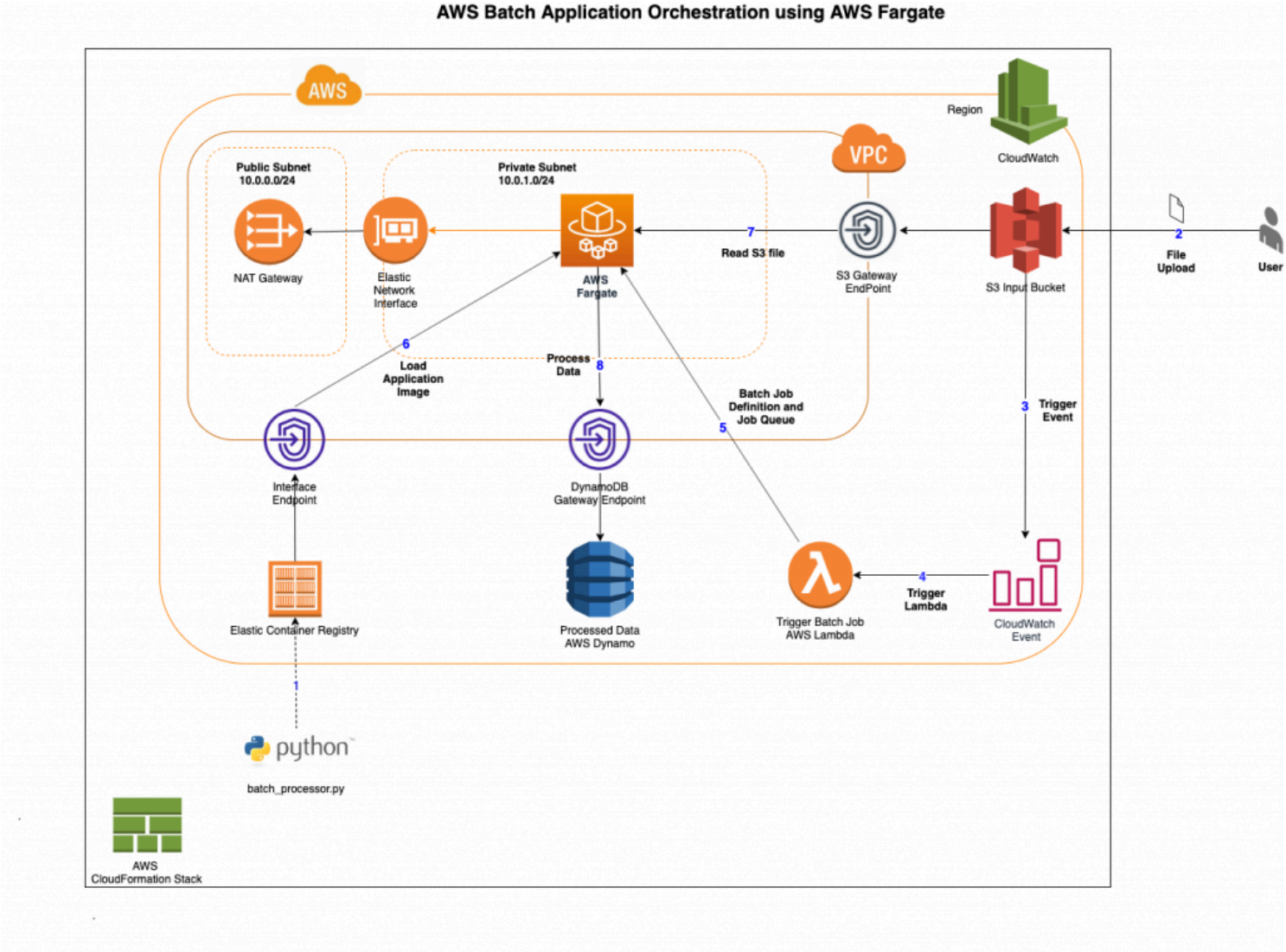
1. Deploying the AWS CloudFormation template – Run the CloudFormation template (command provided) to create the necessary infrastructure.
2. Docker Build and Push – Set up the Docker image for the job:
    1. Build a Docker image.
    2. Tag the build and push the image to the repository.
3. Testing – Drop the CSV into the S3 bucket (copy paste the contents and create them as a `[sample file csv]`). CLI provided to upload the S3 to the created bucket
4. Validation – Confirm that the job runs and performs the operation based on the pushed container image. The job parses the CSV file and adds each row into the DynamoDB table.

Points to consider

- The provided AWS CloudFormation template has all the services (refer to upcoming diagram) needed for this walkthrough in one single template. In an ideal production scenario, you might split them into different templates for easier maintenance.
- To handle a higher volume of CSV file contents, you can do multithreaded or multiprocessing programming to complement the AWS Batch performance scale.
- Solution provided here lets you build, tag, and push the docker image to the repository (created as part of the stack). We've provided both a consolidated script files — `exec.sh` and `cleanup.sh,` as well as include individual commands, should you choose to manually run them to learn the workflow better. You can use the scripts included in this post with your existing CI/CD tooling, such as AWS CodeBuild, or any other equivalent to build from repository and push to AWS ECR.
- The example included in this blog post uses a simple AWS Lambda function to run jobs in AWS Batch, and the Lambda function code is in Python. You can use any of the other programming languages supported by AWS Lambda, for your function code. As an alternative to Lambda function code, you can also use AWS StepFunctions a low-code, visual workflow alternative to initiate the AWS Batch job.

1. Deploying the AWS CloudFormation template

When deployed, the AWS CloudFormation template creates the following infrastructure.



Download the source from the GitHub location. Follow the steps below to use the downloaded code. The `exec.sh` script included in the repo will execute the CloudFormation template spinning up the infrastructure, a Python application (.py file) and a sample CSV file.

Note: `exec_ec2.sh` is also provided for reference. This has the implementation done using Managed EC2 instances as mentioned in the previous blog.

```
$ git clone https://github.com/aws-samples/aws-batch-processing-job-repo
```

```
$ cd aws-batch-processing-job-repo
```

```
$ ./exec.sh
```

Alternatively, you can also run individual commands manually as provided below to setup the infrastructure, push the docker image to ECR, and add sample files to S3 for testing.

Step 1: Setup the infrastructure

```
$ STACK_NAME=fargate-batch-job
```

```
$ aws cloudformation create-stack --stack-name $STACK_NAME --parameters
ParameterKey=StackName,ParameterValue=$STACK_NAME --template-body
file://template/template.yaml --capabilities CAPABILITY_NAMED_IAM
```

After downloading the code, take a moment to review the “templates.yaml” in the “src” folder. The snippets below provide an overview of how the compute environment and a job definition can be specified easily using the managed serverless compute options that were introduced. “templates_ec2.yaml” has the older version of implementation done for EC2 as Compute Environment.

```
ComputeEnvironment:
    Type: AWS::Batch::ComputeEnvironment
    Properties:
      Type: MANAGED
      State: ENABLED
      ComputeResources:
        Type: FARGATE
        MaxvCpus: 40
        Subnets:
        - Ref: PrivateSubnet
        SecurityGroupIds:
        - Ref: SecurityGroup
      ...
```

```
BatchProcessingJobDefinition:
    Type: AWS::Batch::JobDefinition
    Properties:
      ....
      ContainerProperties:
        Image:
        ...
        FargatePlatformConfiguration:
          PlatformVersion: LATEST
        ResourceRequirements:
          - Value: 0.25
            Type: VCPU
          - Value: 512
            Type: MEMORY
        JobRoleArn:  !GetAtt 'BatchTaskExecutionRole.Arn'
        ExecutionRoleArn:  !GetAtt 'BatchTaskExecutionRole.Arn'
        ...
        ...
      PlatformCapabilities:
      - FARGATE
```

When the preceding CloudFormation stack is created successfully, take a moment to identify the major components. The CloudFormation template spins up the following resources, which you can also view in the AWS Management Console.

1. CloudFormation Stack Name – fargate-batch-job
2. S3 Bucket Name – fargate-batch-job<*YourAccountNumber*>
    1. After the sample CSV file is dropped into this bucket, the process should kick start.
3. JobDefinition – BatchJobDefinition
4. JobQueue – fargate-batch-job-queue
5. Lambda – fargate-batch-job-lambda
6. DynamoDB – fargate-batch-job
7. Amazon CloudWatch Log – This is created when the first execution is made.
    1. /aws/batch/job
    2. /aws/lambda/LambdaInvokeFunction
8. CodeCommit – fargate-batch-jobrepo
9. CodeBuild – fargate-batch-jobbuild

Once the above CloudFormation stack creation is complete in your personal account, we need to containerize the sample Python application and push it to ECR. We will use command line execution shell script to deploy the application and the infrastructure. Note that AWS CodeCommit and CodeBuild infrastructure are also created as part of the above template. The template on AWS CodeBuild can also run similar commands to deploy the necessary infrastructure.

2. Docker Build and Push

A simple Python application code is provided (in "src" folder). This is Docker containerized and pushed to the AWS Elastic Container Registry that was created with the CloudFormation template.

```
$ STACK_NAME=fargate-batch-job
$ REGION=$(aws ec2 describe-availability-zones --output text --query 'AvailabilityZone

$ ACCOUNT_NUMBER=$(aws sts get-caller-identity --query 'Account' --output text)

$ SOURCE_REPOSITORY=$PWD
$ docker build -t batch_processor .

$ docker tag batch_processor $(aws sts get-caller-identity --query 'Account' --output
$ aws ecr get-login-password --region $REGION | docker login --username AWS --password
$ docker push $(aws sts get-caller-identity --query 'Account' --output text).dkr.ecr.$
```

3. Testing

In this step we will go over testing with a `sample.csv`, either uploaded manually or via the AWS CLI to the AWS S3 bucket. Make sure to complete the previous step of pushing the built image of the Python code to ECR, before testing. You can verify your build and push by going to the AWS Console > ECR – "fargate-batch-job" repository

1. You will see AWS S3 bucket – fargate-batch-job- is created as part of the stack.
2. you will need to drop the provided `sample.csv` into this S3 bucket. Once you add the file to S3, the Lambda function will be triggered to start AWS Batch. To run the command manually, execute the following —

```
Bash
$ aws s3 --region $REGION cp $SOURCE_REPOSITORY/sample/sample.csv s3://$STACK_NAME-$
```

3. In AWS Console > Batch, notice that the Job runs and performs the operation based on the pushed container image. The job parses the CSV file and adds each row into DynamoDB.

4. Validation

1. In AWS Console > DynamoDB, look for “fargate-batch-job” table. Note sample products provided as part of the CSV is added by the batch

Code cleanup

Use the `cleanup.sh` script to remove the Amazon S3 files, Amazon ECR repository images and the AWS CloudFormation stack that was spun up as part of previous steps.

```
$ ./cleanup.sh
```

Alternatively, you can follow the steps below to manually clean up the built environment

1. AWS Console > S3 bucket – fargate-batch-job- – Delete the contents of the file
2. AWS Console > ECR – fargate-batch-job-repository – delete the image(s) that are pushed to the repository
3. Run the below command to delete the stack.

```
$ aws cloudformation delete-stack --stack-name fargate-batch-job
```

1. To perform all the above steps in CLI

```
$ SOURCE_REPOSITORY=$PWD
$ STACK_NAME=fargate-batch-job
$ REGION=$(aws ec2 describe-availability-zones --output text —query 'AvailabilityZones

$ ACCOUNT_NUMBER=$(aws sts get-caller-identity --query 'Account' —output text)

$ aws ecr batch-delete-image --repository-name $STACK_NAME-repository --image-ids imag
$ aws ecr batch-delete-image --repository-name $STACK_NAME-repository --image-ids imag
$ aws s3 --region $REGION rm s3://$STACK_NAME-$ACCOUNT_NUMBER —recursive
$ aws cloudformation delete-stack --stack-name $STACK_NAME
```

References

1. New – Fully Serverless Batch Computing with AWS Batch Support for AWS Fargate
2. AWS Batch on AWS Fargate
3. Manage AWS Batch with Step Functions

Conclusion

Following the solution in the post, you will be able to launch an application workflow using AWS Batch integrating with various AWS services. You also have access to the Python script, CloudFormation template, and the sample CSV file in the corresponding GitHub repo that takes care of all the preceding CLI arguments for you to build out the job definitions.

As stated in the post, with AWS Fargate, you no longer have to provision, configure, or scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale your clusters, or optimize cluster packing. With Fargate or Fargate Spot, you don't need to worry about Amazon EC2 instances or Amazon Machine Images. Just set Fargate or Fargate Spot, your subnets, and the maximum total vCPU of the jobs running in the compute environment, and you have a ready-to-go Fargate computing environment. With Fargate Spot, you can take advantage of up to 70% discount for your fault-tolerant, time-flexible jobs.

I encourage you to test this example and see for yourself how this overall orchestration works with AWS Batch. Then, it is just a matter of replacing your Python (or any other programming language framework) code, packaging it as a Docker container, and letting the AWS Batch handle the process efficiently.

If you decide to give it a try, have any doubt, or want to let me know what you think about the post, please leave a comment!

About the Author

Sivasubramanian Ramani (Siva Ramani) is a Sr Cloud Application Architect at AWS. His expertise is in application optimization, serverless solutions with AWS.

TAGS: AWS Batch

Comments

0 Comments

Start the discussion...

♡ 2 **Share**