FastAPI Benchmark and Deployment Report

FastAPI Deployment Procedure

Key Pair and Security Group

The first step in deploying the FastAPI application was **generating an SSH key pair**. This key is essential for securely accessing the EC2 instances. The command `generate_key_pair` was used to create the key, allowing us to securely SSH into the instances.

Additionally, we created a security group (`create_security_group`) to manage the inbound and outbound traffic for our instances. The security group was configured to allow:

- SSH (Port 22) for remote access.
- HTTP/HTTPS (Ports 80, 443) for web traffic.

Worker Instances Setup

For each worker, we used `get_user_data`, which provides a script that installs essential packages such as Python3, FastAPI, and Uvicorn. The worker instances serve different FastAPI endpoints (*/cluster1*, */cluster2*) and are part of the cluster used for handling traffic.

Each worker runs the FastAPI application via Uvicorn. The load balancing and clustering were achieved by tagging these instances and distributing the requests based on their CPU load.

Load Balancer Deployment

The load balancer was set up using `get_lb_user_data`. This script installs dependencies and runs a Python script (`load_balancer.py`) on a dedicated instance to handle traffic distribution between workers. The load balancer:

- Uses CloudWatch metrics to monitor CPU utilization of each worker.
- Distributes incoming requests to the worker with the lowest CPU usage.

Cluster Setup Using Application Load Balancer

We deployed two clusters:

- **Cluster 0:** Instances of type t2.micro.
- **Cluster 1:** Instances of type t2.large.

Each cluster was tagged accordingly, allowing the load balancer to differentiate between them. The load balancer was configured to route traffic between these clusters based on CPU utilization metrics.

Tagging and Instance Management

Each instance was tagged as part of either Cluster 0 or Cluster 1. The load balancer uses these tags to direct traffic, ensuring that requests are handled efficiently. The custom logic for this was written in Python and deployed on an EC2 instance running the load balancer.

Benchmark Results

Expectations

Before any experiment, we expect that the t2.micro cluster reaches its limits (CPU utilization, Latency and Response Time) before the t2.large cluster. Indeed, the capacity of the virtual machines entail different behaviors regarding the amount of requests.

CPU Utilization

Latency and Response Times

We tested the latency and response times for each cluster by sending 1000 requests to the load balancer and observing how long each cluster took to respond.

- **t2.micro :**
- **t2.large :**

Throughput

Instructions to Run code

Prerequisites

To run the FastAPI deployment and benchmark tests, the following prerequisites must be met:

- AWS account: Ensure you have an active AWS account with appropriate IAM roles to create EC2 instances and security groups.

- Python 3: Install Python 3.x along with the boto3 library for interacting with AWS services.
- AWS CLI: Install the AWS CLI and configure your credentials with the aws configure command.

EC2 Instance Creation

Run the `main.py` script to:

1. Generate the key pair.
2. Create the security group.
3. Launch the EC2 instances (workers and load balancer).

Ensure your AWS credentials are either set up in the environment or passed directly to the script.

Deploy FastAPI

Once the instances are up, the FastAPI workers will automatically start. You can verify this by sending HTTP requests to the */cluster1* or */cluster2* endpoints of the workers. For example:

```
curl http://<worker_public_ip>:8000/cluster1
```

Test tke Load Balancer

To test the load balancer, send requests to its public IP. The load balancer will distribute the traffic to the worker instances based on their current CPU utilization:

```
curl http://<load_balancer_public_ip>
```

Conclusion

In this project, we successfully deployed a FastAPI application on AWS EC2 instances, setting up a custom load balancer to manage traffic between two clusters. The performance benchmarks demonstrated the clear advantage of using t2.large instances for high-traffic applications, though t2.micro instances can be useful for low-traffic scenarios.

While the custom load balancer provided granular control over traffic distribution, AWS's managed Elastic Load Balancer (ELB) may offer a more robust and scalable solution for production environments, particularly with features like SSL termination and automatic failover.