FastAPI Benchmark and Deployment Report

Authors : Stéphane Michaud 1904016 - Stephen Cohen 2412336 - Zabiullah Shair Zaie 2087651 - Asma Boukhdhir 2412257

- FastAPI Benchmark and Deployment Report
  - Authors : Stéphane Michaud 1904016 - Stephen Cohen 2412336 - Zabiullah Shair Zaie 2087651 - Asma Boukhdhir 2412257
  - FastAPI Deployment Procedure
    - Key Pair and Security Group
    - Worker Instances Setup
    - Load Balancer Deployment
  - Cluster Setup Using Application Load Balancer
    - Tagging and Instance Management
    - Expectations
  - Benchmark Results
  - Instructions to Run code
    - Prerequisites
  - EC2 Instance Creation
    - Deploy FastAPI
    - Test the Load Balancer
  - Conclusion
  - Appendix
    - Code
    - logs :

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 responsiveness of http requests to determine the cluster with the least load.
- Distributes requests between the clusters based on their responsiveness.
- This responsiveness is calculated after a fix number of requests sent. (10 on each cluster in Folder load_balancer : in load_balancer.py)

Cluster Setup Using Application Load Balancer

We deployed two clusters:

- **Cluster 1:** Instances of type t2.micro.
- **Cluster 2:** 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 the time response of instances. The fastest to answer will have the request.

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.

Expectations

Before any experiment, we expect that the t2.micro cluster to have higher response time than the t2.large. It is to note however that the workload is not expensive, so the difference in response time might not be significant. Indeed, the capacity of the virtual machines entail different behaviors regarding the amount of requests.

Benchmark Results

The raw results of the benchmark are the following :

- Cluster 1 (t2.micro) :

```
Total time taken: 6.75 seconds
Average time per request: 0.0067 seconds
```

- Cluster 2 (t2.large) :

```
Total time taken: 5.87 seconds
Average time per request: 0.0059 seconds
```

We conclude that t2.large cluster are 14% more efficient than t2.micro cluster. (in terms of time response). This follows the feelings we have, that t2.large cluster handled easier the requests.

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).
4. Run the benchmark
5. Clean-up all the instances, the key-pair generated and the security-group

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 the 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.

Appendix

Code

All the code is available in Github.

logs :

All the logs are available in : benchmark_log.txt