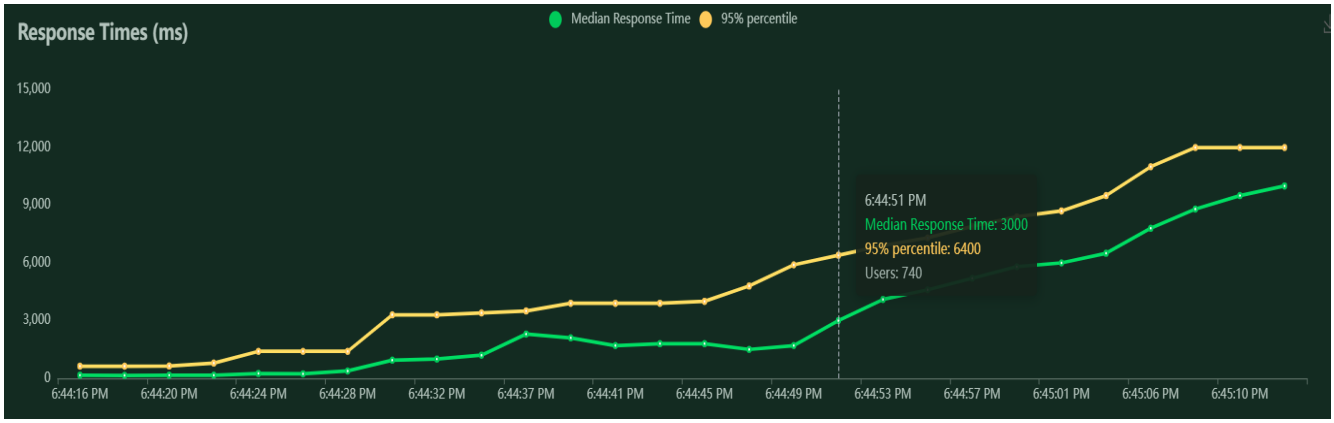
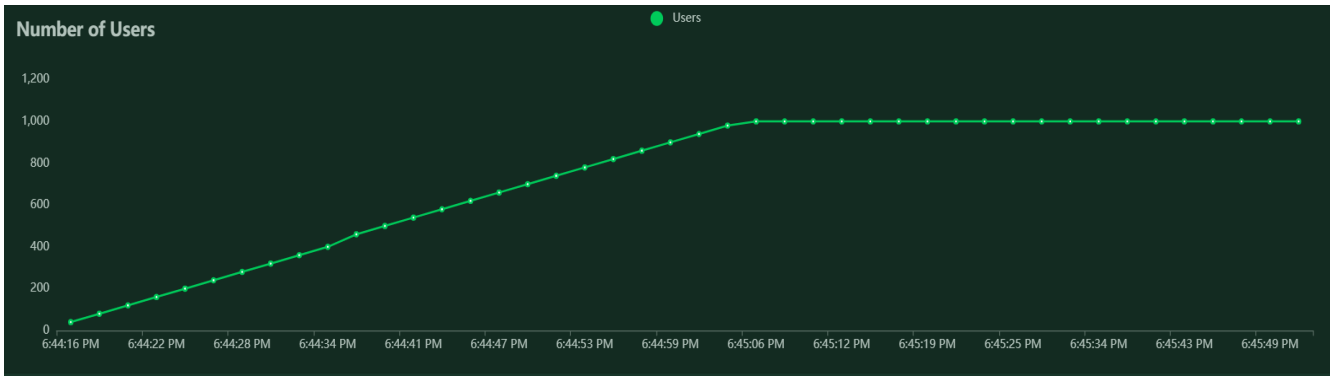
Cloud-based scalability design for WolfComplain2.0

We perform load testing using Locust where we can simulate customer traffic, by means of swarming our hosted service, with many requests per second to see how well our auto-scaling functionality works. We consider faster response times as a parameter to measure scalability





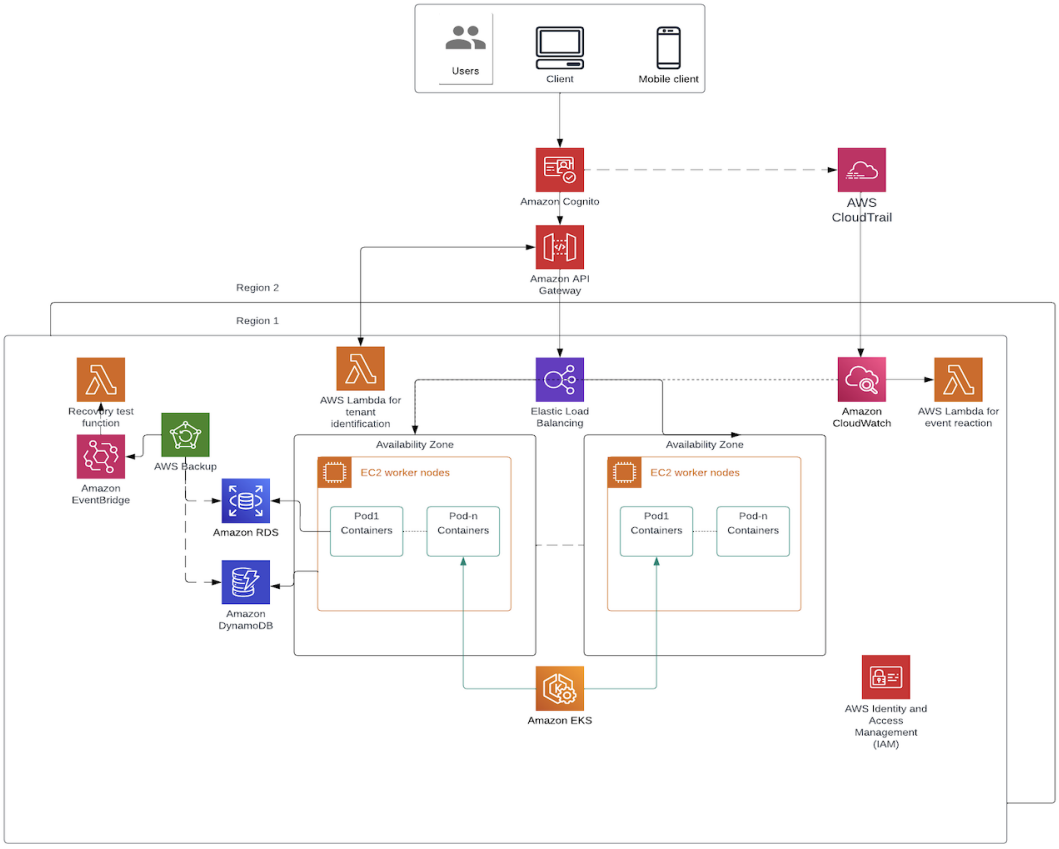
From the graphs, we can observe that WolfComplain has a response time of 3000 ms for 1000 user requests.

Scalability

To implement scalability, we plan to host the application on AWS. We propose the following architecture to ensure that the application responds to time-varying loads. The kubernetes metric server is used to stream the device metric to the kubernetes **horizontal pod autoscalar**, which scales up/down the pods based on the CPU utilization metric of the pods. The parameters to be configured for the autoscaling are MINPODS and MAXPODS.

Horizontal scaling means that the response to increased load is to deploy more pods. The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:
desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

For example, if the current metric value is 200m, and the desired value is 100m, the number of replicas will be doubled, since 200.0 / 100.0 == 2.0 If the current value is instead 50m, you'll halve the number of replicas, since 50.0 / 100.0 == 0.5. The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default). When a targetAverageValue or targetAverageUtilization is specified, the currentMetricValue is computed by taking the average of the given metric across all Pods in the HorizontalPodAutoscaler's scale target **[1]**



The clients send requests with the login credentials to **Amazon Cognito** to authenticate and access the application, Amazon Cognito authenticates the user based on the credentials. Upon successful authentication, Cognito determines the role of the user and classifies the tenant along with **Amazon API Gateway**.
The **Elastic Load Balancers (ELBs)** automatically distributes the traffic across multiple targets such as EC2 instances and containers in one or more Availability Zones (AZs). It routes traffic to

only healthy targets and scales in response to the incoming traffic. We choose application load balancers to load balance HTTP requests

The **Amazon EKS** service has the sole responsibility of ensuring that the cluster has sufficient EC2 worker nodes in order to schedule pods and not waste resources.

1. Once the load increases more than the percentage that can be handled by a single pod, the pod goes into a Pending State.
2. The Kubernetes cluster is monitoring the pod and realizes that the pod is unable to handle the requests which are received by it since the EC2 worker node cannot have any new pods.
3. The EKS then simulates the addition of nodes by increasing the number of instances in the particular Autoscaling Group.
4. This in turn provisions the new EC2 worker node that can handle this increased number of requests.
5. The new EC2 worker node schedules a pod within it.
6. EKS then monitors the pods and watches for any pod which fails to schedule and also for nodes which are underutilized, so that it can again make decisions to either scale up or down.

The microservices performs varied database access for read/write operations and the design leverages on various amazon services to cater to the needs. The services which the microservices interacts with are the following:

- DynamoDB
- RDS

In order to avoid single-point-of-failure, redundancy is ensured by using two Availability Zones (AZ) across different regions. We propose a recovery management system using **Cloudwatch** where the alarms for resources are monitored and recovery procedures are triggered in case of any events to restore the resources from the backup.

Reference:

[1] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/