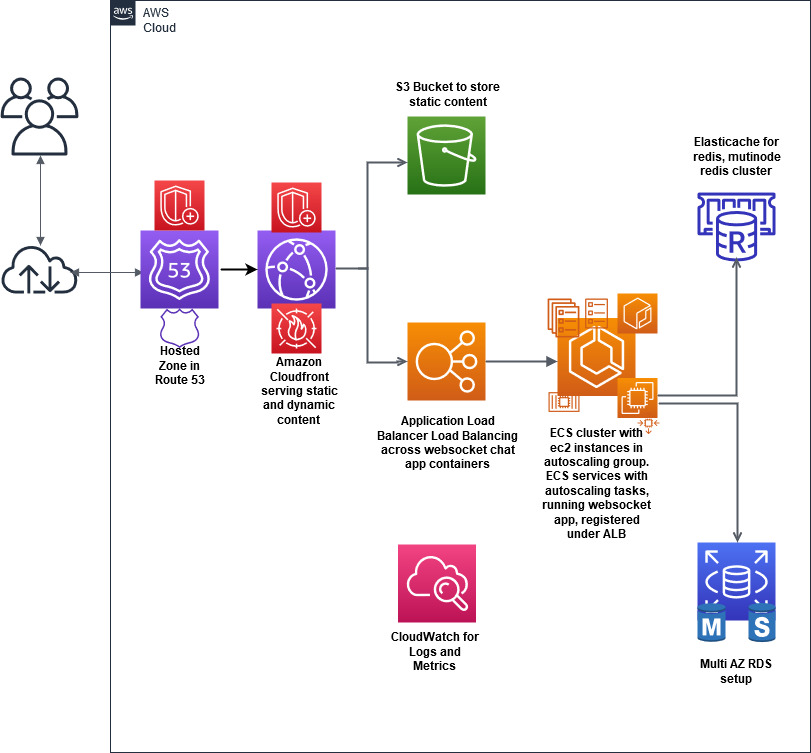
Based on the requirement, I have come up with the following designs for implementing a highly available, scalable, secured infrastructure on AWS.

Solution 1 –



Using sort of as-is architecture as the on prem setup mentioned in the requirement, this implementation has

**A hosted zone on Route53**, which is going to provide the DNS resolution of our app url. There is a type A record set, the alias of which is mapped to Domain name of CloudFront Distribution serving the website.

**The CloudFront Distribution** has two origins, one mapping to the S3 bucket serving the static content and this is cached. Another origin points to the Application Load Balancer for /app path pattern, which is not cached. Serves all web urls over HTTPS. Certificate generation and management in Amazon Certificate Manager.

**VPC Setup**

VPC with 28.9.0.0/16 CIDR range

**Subnets**

There are 3 public subnets across three availability zones, where the Application Load Balancer is going to be setup.

There are 3 private subnets across three availability zones for compute resources like the EC2 launched in ECS cluster.

There are 3 private subnets across three availability zones for DB resources like AWS RDS Maria DB instances and Elasticache Redis instances.

A Nat Gateway will be attached to the 6 private subnets.

**Security Groups**

Security Group for Application Load Balancer – Allows port 80 for all the CloudFront IPs. If the IPs change, a lambda function setup and subscribed to AmazonSpaceIPChanged SNS topic, will update the security group with correct latest IPs. As mentioned here - https://aws.amazon.com/blogs/security/how-to-automatically-update-your-security-groups-for-amazon-cloudfront-and-aws-waf-by-using-aws-lambda/

Security Group for EC2 instances in ECS cluster – Allow the ephemeral host port 32768 – 61000 from the ALB security group, which are used when containers are mapped to dynamic host ports. This allow to run multiple same tasks on same ec2 instance which helps in quick autoscaling.

Security group for RDS instances – For MariaDB RDS instances, allow port 3306 from the private compute subnets.

**ECS cluster**

For running the nodejs or java websocket chat app, we are going to run them as container or tasks in an ECS cluster consisting of ec2 instances in an autoscaling group. Tasks will be run by a Service which is attached to the Application Load balancer’s target group. The Service will have desired count of tasks as 2. Autoscaling policy will be setup in the ECS service, where based on CPU utilization of more than 70 percent, tasks will be scaled out to maximum as specified. Also, tasks will be scaled down if the cpu utilization is less than 30 percent. These values are taken as reference for this demo purpose and can be reconfigures easily as required.

**Application Load Balancer**

An Application Load balancer will load balance across the tasks in ECS service. It will be listening to HTTP requests on port 80, from the Cloudfront IPs and serve dynamic content via Cloudfront Distribution. Currently it will serve the /api requests via one target group. It supports Websocket based communication.

**RDS Instance**

A MultiAZ MariaDB RDS instance will be setup. Data Encrypted at rest using KMS keys. Data in transit can be encrypted using a SSL connection.

**Elasticache for Redis**

A 2 node MultiAZ Elasticache Redis implementation will be used for this solution. Encryption at rest and transit must be setup, though not implemented in the actual demo.

**S3 Buckets**

A bucket for keeping the static contents.

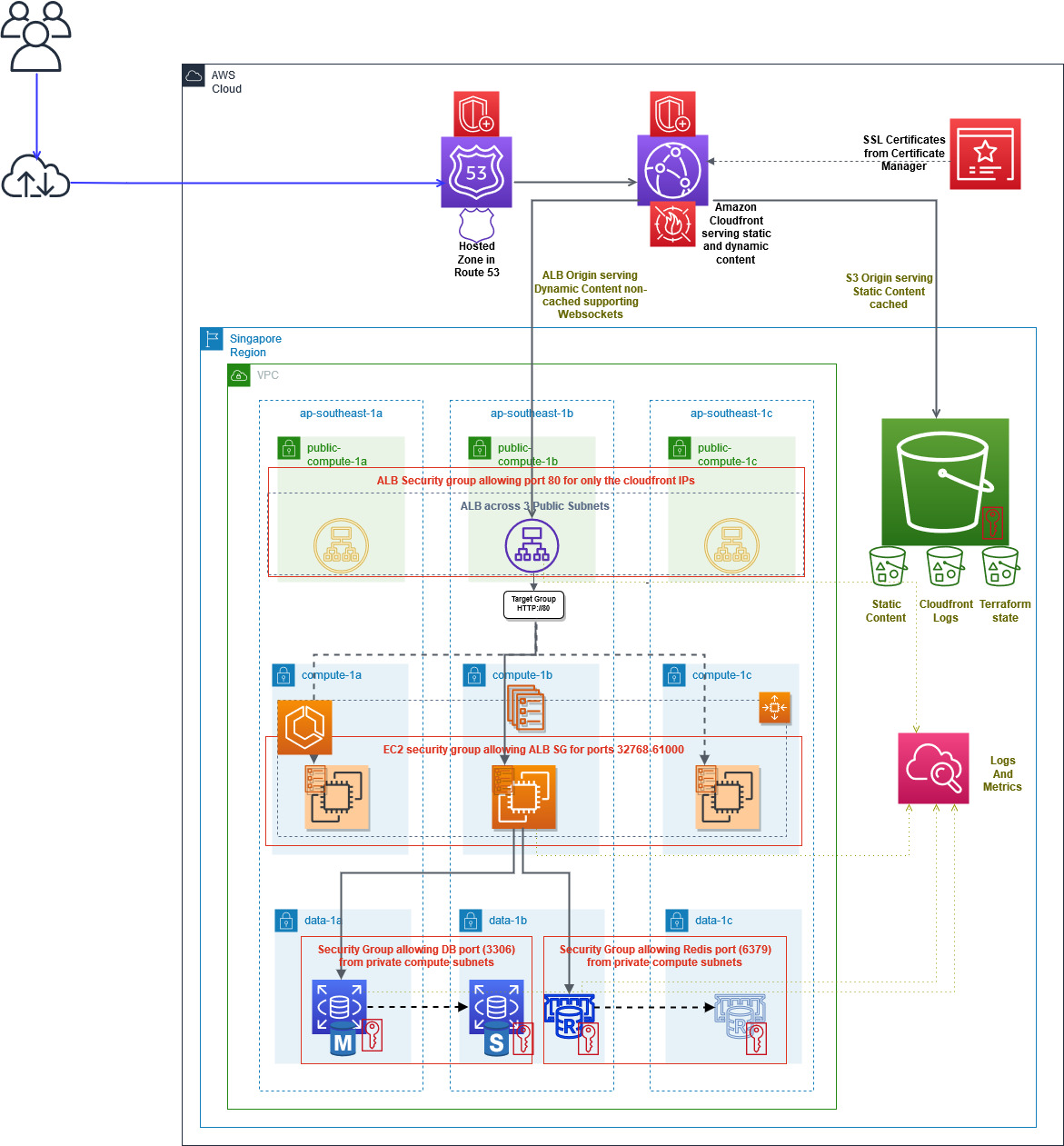
One bucket for Cloudfront logs

One bucket for storing the terraform state files.

**Cloudwatch**

Metrics and logs captured from most of the services used.

**AWS Design Solution 1 Detailed Diagram**

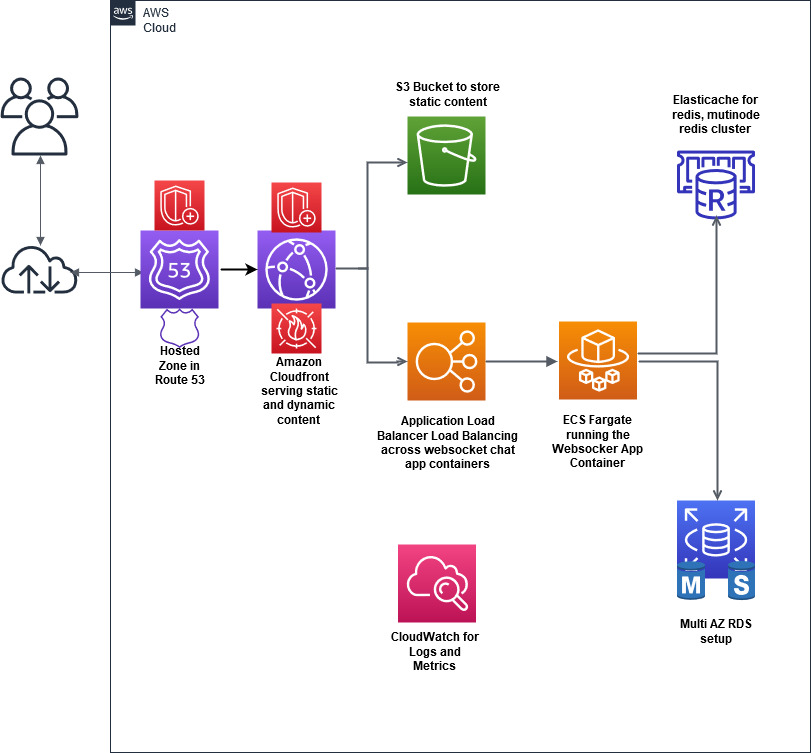


draw.io xml files for the above solution



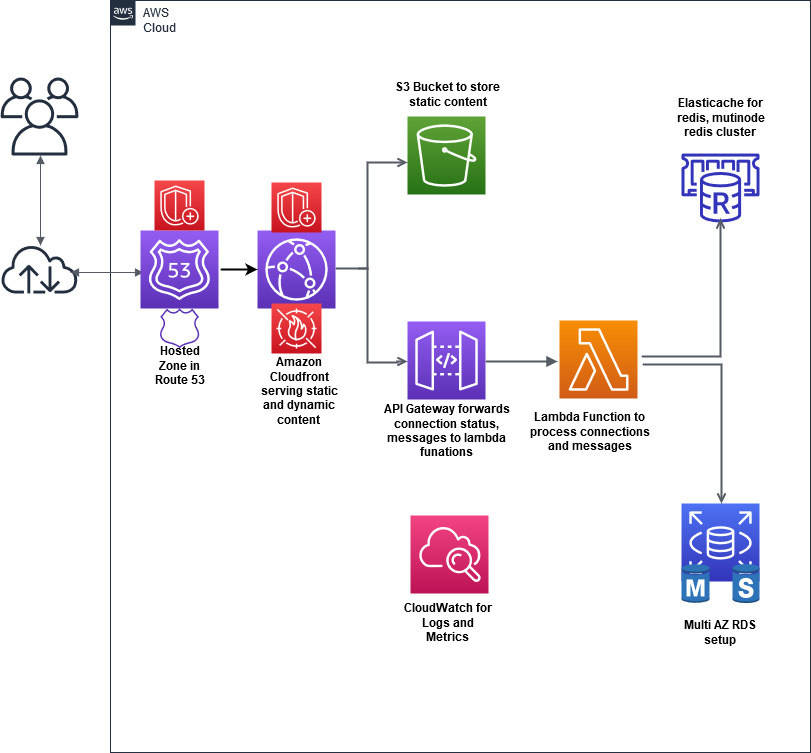
**Solution 2**

Moving towards serverless, the dynamic content served by ECS fargate tasks, instead of tasks running on EC2.



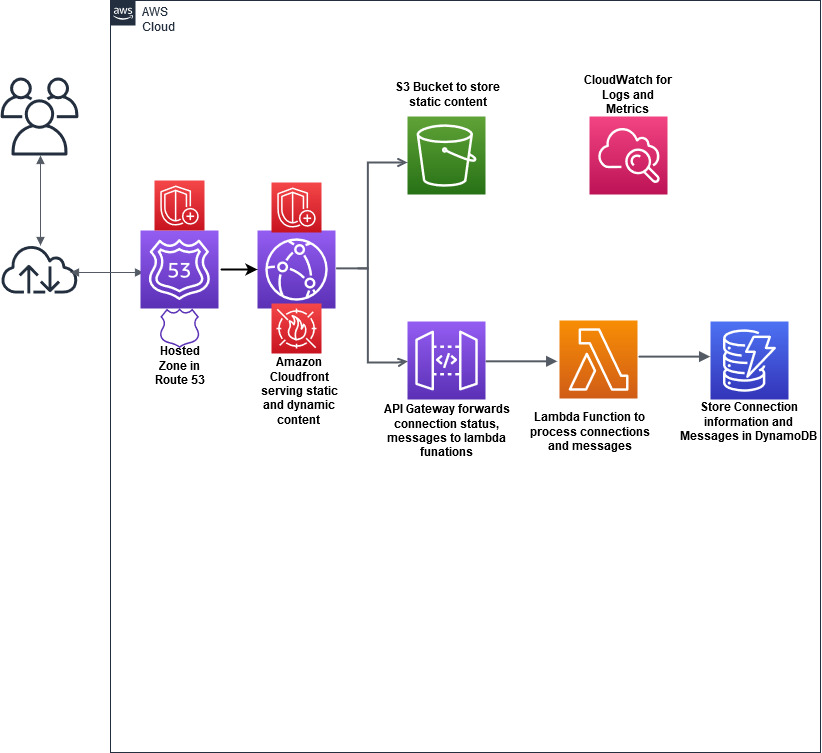
**Solution 3**

Instead of application load balancer and ECS, this solution uses API Gateway and Lambda Functions to serve dynamic content. API Gateway has Websocket support.



**Solution 4**

Further simplifying Solution 3, RDS and Redis Pub/Sub replaced by DynamoDB



**Solution 5**

Using Appsync to build a websocket chat application as mentioned here –

Building a serverless real-time chat application with AWS AppSync

<https://aws.amazon.com/blogs/mobile/building-a-serverless-real-time-chat-application-with-aws-appsync/>

