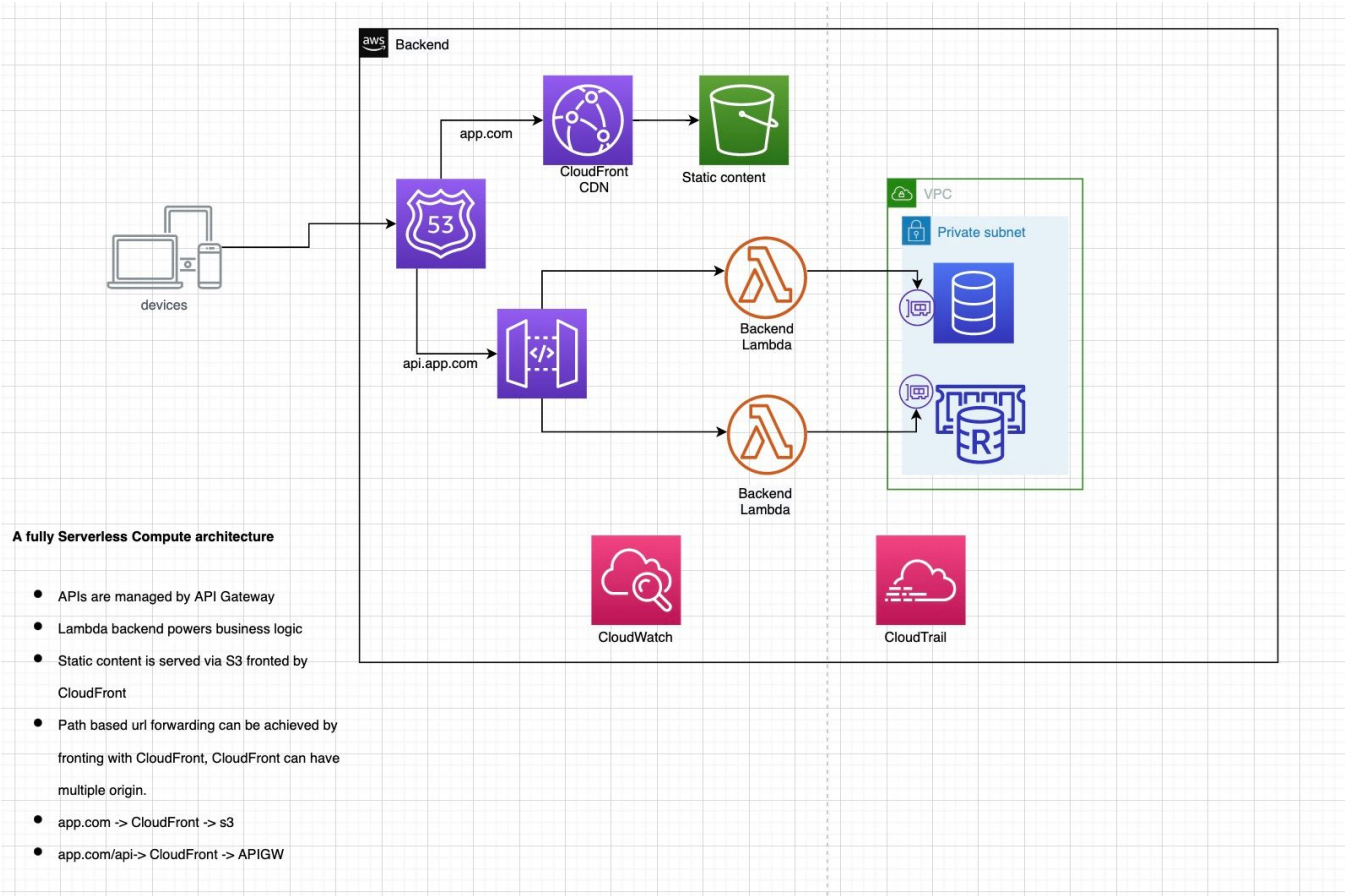
Migration to AWS

- For the given use case, multiple options are possible depending on how the app is already written and deployed. For example, if app rewrite overhead is allowed, deploying on a serverless Function as a Service product like AWS Lambda will be the best choice. Else if application is deployed as multiple microservices on a container runtime like docker, service like AWS Elastic Container Service can be preferred. If App need long running fully managed on-prem server like system, AWS EC2 along with Auto Scaling Group(ASG) can serve the compute requirements.
- With this background, three architectures can be considered:

Fully Serverless Computer Architecture

Design

- AWS Api Gatewayv2 supports websocket API that can handle bidirectional communication and manage connections. This Service can be used to create and manage API part of the application. The service is fully managed by AWS and runs on high availability compute is highly scalable. ApiGateway also comes with other features like throttling/caching.
- Backend functionality can be deployed to another AWS serverless product, AWS Lambda. Lambda is also fully managed by AWS and handles high availability requirements
- For storage, RDS and ElasticCache(redis) setup can server application requirements. RDS comes with a multi-AZ mode with a stand-by instance for auto failover and supports Read replicas. RDS now also supports storage AutoScaling to cater increasing storage needs. ElastiCache also comes with auto-failover and multi-AZ support. Both these clusters, run in a private subnet and are accessed by lambda which runs the application logic.
- Static Content can be served by CloudFront which is AWS CDN product with s3 holding the actual content. Allowing s3 bucket public access is not a security best practice and hence fronting with CloudFront gives a good security posture plus CDN advantages..



- Logs are written to CloudWatch which is a logging service by AWS and also API actions are logged by CloudTrail.
- Route53(AWS DNS service) can be used to create a hosted zone (app.com) which can have Resource Records aliasing to CloudFront and ApiGW public HTTP DNS.

Note: *The above architecture has 2 domains app.com and subdomain api.app.com serving static and API content. I've missed the question statement that we need one domain name with different routes. This can be achieved by fronting app with CloudFront . CloudFront supports multi-origin and can route to different origins based on request*

app.com Cloudfront S3

app.com/apiCloudfrontAPIGW(publically accessible Https endpoint)

Implementation

- The above Design is implemented with a sample chat application. The application source is written in Python which is deployed as Lambda Functions. These lambda functions are fronted by AWS Websocket API which handles connections.
- Redis is used to store connection information of logged users and RDS holds a list of pre-registered users who are allowed to communicate.
- Infrastructure as code is implemented in Pulumi Python SDK.
- The API functionality is tested and works all good, and the code is hosted at https://github.com/skmamillapalli/pulumi-aws-chatapp

```
Previewing update (dev):
     Type                              Name                            Plan
 +   pulumi:pulumi:Stack               chatapp_pulumi_iac-dev          create
 +   ├─ aws:ec2:Eip                    eip1                            create
 +   ├─ aws:ec2:Vpc                    chatapp-vpc                     create
 +   ├─ aws:iam:Role                   sendMessagelambda               create
 +   ├─ aws:apigatewayv2:Api           ChatAppApi                      create
 +   ├─ aws:lambda:LayerVersion        helper-layers                   create
 +   ├─ aws:ec2:Subnet                 PublicSubnet                    create
 +   ├─ aws:ec2:Subnet                 PrivateSubnet1                  create
 +   ├─ aws:ec2:Subnet                 PrivateSubnet2                  create
 +   ├─ aws:ec2:InternetGateway        inet-gateway                    create
 +   ├─ aws:ec2:SecurityGroup          lambdaSG                        create
 +   ├─ aws:iam:RolePolicy             RolePolicyAttachment            create
 +   ├─ aws:apigatewayv2:Stage         Devstage                        create
 +   ├─ aws:ec2:NatGateway             nat-gateway                     create
 +   ├─ aws:elasticache:SubnetGroup    RedisSubGroup                   create
 +   ├─ aws:rds:SubnetGroup            rdssubgroup                     create
 +   ├─ aws:ec2:RouteTable             publicsubnetroutetable          create
 +   ├─ aws:ec2:SecurityGroup          AllowLambdaToRdsIngress         create
 +   ├─ aws:ec2:SecurityGroup          AllowLambdaToRedisIngress       create
 +   ├─ aws:ec2:RouteTable             privatesubnetroutetable         create
 +   ├─ aws:elasticache:Cluster        redisnode                       create
 +   ├─ aws:rds:Instance               default                         create
 +   ├─ aws:ec2:RouteTableAssociation  PublicSubnetRT                  create
 +   ├─ aws:ec2:RouteTableAssociation  PrivateSubnetRT1                create
 +   ├─ aws:ec2:RouteTableAssociation  PrivateSubnetRT2                create
 +   ├─ aws:lambda:Function            disconnectfunction              create
 +   ├─ aws:lambda:Function            sendmessagefunction             create
 +   ├─ aws:lambda:Function            connectfunction                 create
 +   ├─ aws:lambda:Permission          lambdainvocationpermissions-2   create
 +   ├─ aws:apigatewayv2:Integration   disconnectroute                 create
 +   ├─ aws:lambda:Permission          lambdainvocationpermissions-1   create
 +   ├─ aws:apigatewayv2:Integration   connectroute                    create
 +   ├─ aws:apigatewayv2:Integration   sendmessageroute                create
 +   ├─ aws:lambda:Permission          lambdainvocationpermissions     create
 +   ├─ aws:apigatewayv2:Route         disconnect-route                create
 +   ├─ aws:apigatewayv2:Route         connect-route                   create
 +   └─ aws:apigatewayv2:Route         sendmessage-route               create
```

```
~>wscat -c wss://dmkqfk0fpc.execute-api.ap-southeast-1.amazonaws.com/Dev -H "username:Sunil"
Connected (press CTRL+C to quit)
> {"action":"send", "to":"Sunil", "message":"Hey!!"}
< "Hey!!"
> {"action":"send", "to":"Sid", "message":"Hey!!"}
>
```

*What is **not** implemented and To-do*

- *For the above App, Static Content integration is not tested. i.e from CDN->S3*
- *End to End integration with a full working DNS zone is not tested.*
- *Custom Alarms and Customer Managed keys are a best practice to manage monitoring alarms. This is still backlog.*

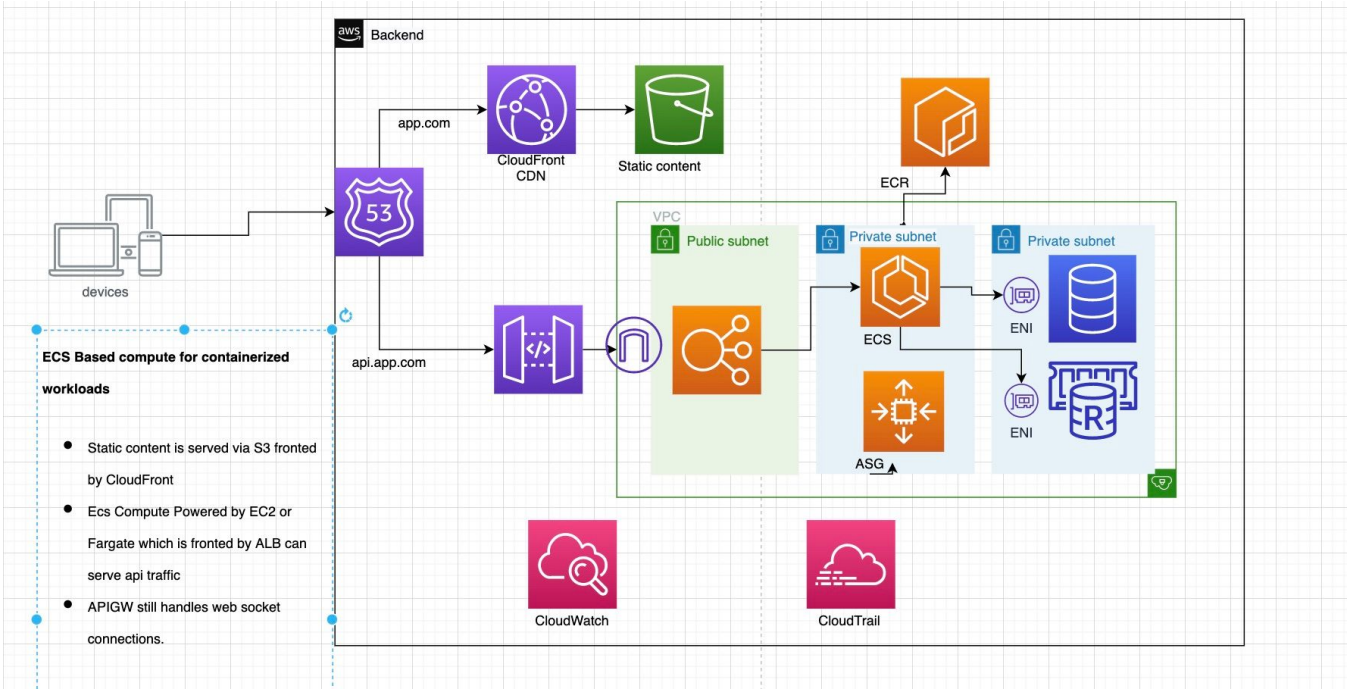
Considerations

- Rewriting applications to functions could be an overhead for App Teams.

ECS Based Compute for Container workloads

Design

- For container workloads, AWS Elastic Container Service can be a potential choice of migration. ECS Cluster can be deployed on managed EC2 or Serverless Fargate infrastructure. ECS Service can be fronted with an ALB that sends to traffic to multiple tasks of a service.
- ECS Service can be deployed multi-az setup and is also integrated with AutoScaling to scale tasks based on CloudWatch metrics.



- Api Gateway is still the app fronting part given the flexibility it offers with handling web sockets. Since ALB is in Public Subnet, APIGW can be tied with a HTTP PROXY integration.
- Storage design is same as the first architecture. RDS and redis clusters handle storage requirements. CloudWatch and Trail are ditto too.

Implementation

- Implementation is **not** done for this architecture

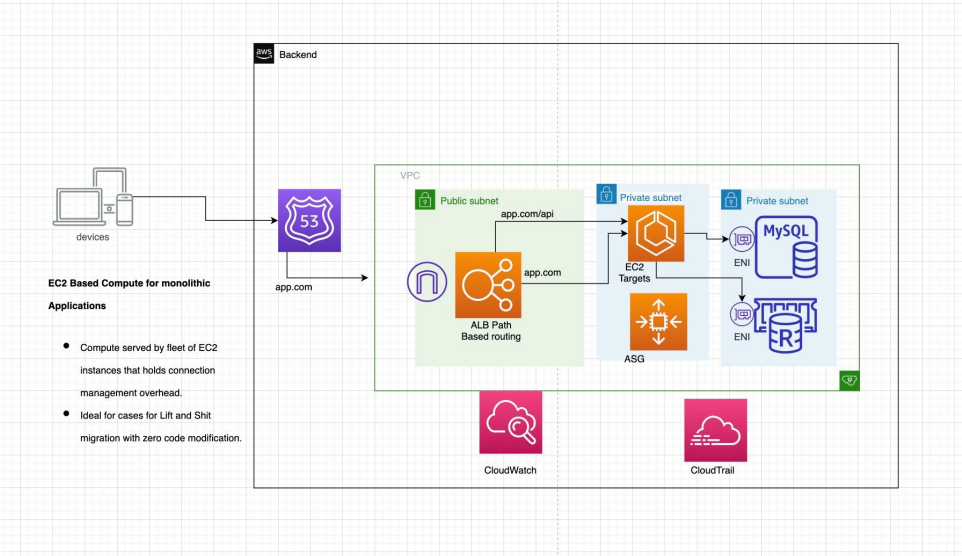
Considerations

- Micro service based applications which run on container runtimes are best candidates for this architecture.

EC2 Instance Based Compute with Auto Scaling

Design

- This design can be good for cases with little or no migration overhead. The setup basically has a fleet of Ec2 instances that fully manages connections and application.
- Auto Scaling Group offers scaling requirements and can be deployed in multi-AZ mode.



- Storage is still the same as previous architectures.
- Application Load Balancer(ALB) is a L7 load balancer that can do path based routing to different targets. So, app.com and app.com/api can be routed to different target groups by ALB.

Implementation

- Implementation is **not** done for this architecture

Data Migration

- For moving on-prem data to AWS, again its a choice based on how much data the app currently holds. AWS has number of services to serve this starting with copying to s3 over http or use a dedicated service like AWS Data Migration Service. AWS also offers snow* products for data transfers not feasible over network.