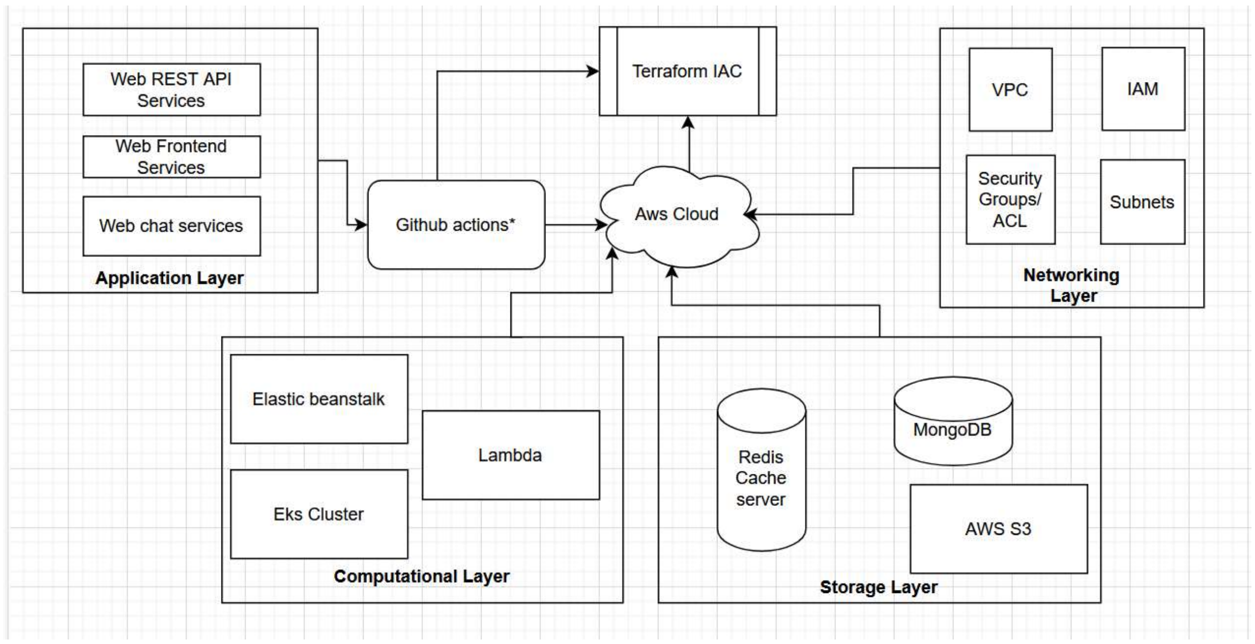
Task2 Assessment

Part 1:

Design the web application: Sketch a high-level architecture diagram of a web application that demonstrates your ability to create a secure and scalable system. Include components like the front end, back end, databases, storage, caching, and networking elements.

Solution:

Overview: N layer Diagram

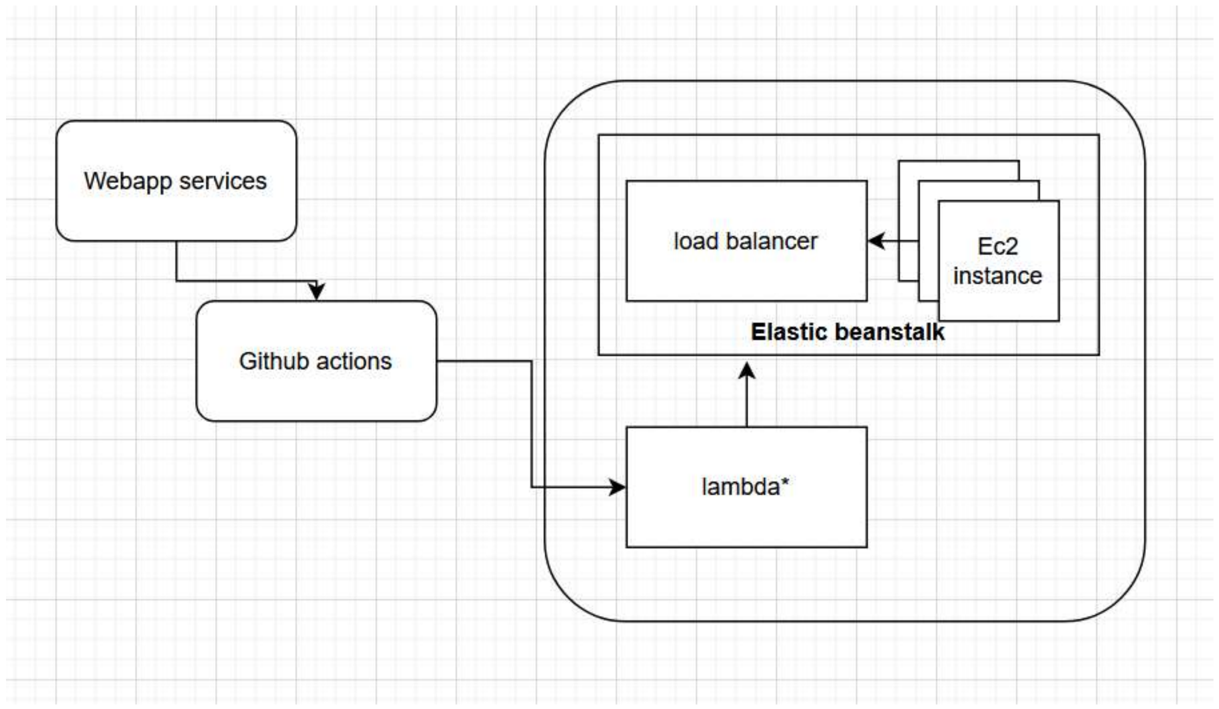


For reference detailed diagram

*Github actions: It can trigger terraform IAC to deploy infrastructure and on success, it will configure and deploy apps containerization services (docker container) on Elastic beanstalk or run bash script for eks cluster.

For scalabilty:

We can use as an example of deploying web app on elastic beanstalk



For reference Lambda*: Based on the traffic and web app requirement, we can trigger lambda function to see the idle time and traffic, and can manually scale elastic beanstalk ec2 instances.

Part 2:

Infrastructure as Code (IaC): Outline the infrastructure setup using Infrastructure as Code concepts, using tools like Terraform, AWS CloudFormation, or similar.

Solution:

For Terraform template:

1. Select the suitable infrastructure technologies based on apps size and computation (traffic, critical importance in term of availability , downtime), e.g for small serverless app use lambda, elastic beanstalk
2. After selection, Divide the infrastructure into reusable modules,
   Example directory structure would be

.

```
├── ├── modules/
├── │   ├── ec2/
├── │   │   ├── main.tf
├── │   │   ├── variables.tf
├── │   │   └── outputs.tf
├── │   ├── s3/
├── │   │   ├── main.tf
├── │   │   ├── variables.tf
├── │   │   └── outputs.tf
├── │   ├── vpc/
├── │   │   ├── main.tf
├── │   │   ├── variables.tf
├── │   │   └── outputs.tf
├── │   └── rds/
├── │      ├── main.tf
├── │      ├── variables.tf
├── │      └── outputs.tf
├── ├── environments/
├── │   ├── dev/
├── │   │   └── main.tf
├── │   ├── staging/
├── │   │   └── main.tf
├── │   └── prod/
├── │      └── main.tf
├── ├── terraform.tfvars
├── ├── provider.tf
├── ├── versions.tf
```

Folder Structure diagram 2.1

3. For staging and production, use separate code or environment to install resources
4. Use IAM, Policies, aws secret manager, Security groups and Access control list resource for ultimate security

Part3:

Cost optimization: Briefly discuss your strategy for optimizing resource usage and costs without compromising security and scalability.

Solution

For cost optimization without interruption and security, it will need strategic decision with strictly follow rules

Some of the good practices and experiences are

1. Use Right instance type or resource based on workload requirements with auto scaling, We have done tremendous research on our expected traffic and resource constraints and deploy Dopel on appservices, elastic beanstalk
2. Based on apps availability and critical plan user reserved instance or saving sharing plan, spot instance was more cost effective in our case but for less critical applications, we can opt shared instance
3. For static data, use s3 glacier for infrequent data and implement data lifecycle policies. For old archive data, we have deployed the data policies where non frequent data will be passed to the glacier and then purge the non useable data after a span of six months.
4. Mitigate data transfer charges by keep resource within a same region or availability zone, Strictly adhere the devops engineer to documented the deployment.
5. Use aws cost explorer to monitor the pattern of spending and make decision to improve cost, Personally reduce cost by almost 45 percent by remove useless services, ebs snapshots, redis servers, remove resource groups by checking cost explorer every fortnight to ensure cost effectiveness
6. Regularly make audits and remove EBS volume or snapshots of resources
7. For small apps use aws lambda, or serverless apps, use cloud front for static sites.