Merapar Tech Test Submission

Solution

The solution I ultimately put together is a small web application, written in Go, that uses a third-party library to produce a random, 10-character string. This string is then injected into HTML using templating. The application is then compiled and executed by the host. Go was chosen because available libraries make it easy to quickly put together a small application.

The application source is pulled from the Git repository during deployment, and is finally compiled and executed on the host.

The application is then hosted in AWS on an EC2 Instance described in Terraform. The application sits within a public subnet and is thus assigned a public IP address. Doing so meets the test criteria but does not consider any other typical, real-life requirements. The solution could also just as easily been a serverless one, using API Gateway and Lambda, with IAC in AWS SAM.

Nonetheless, an EC2 Security Group is attached to the Instance which only permits traffic:

- Ingress 8080/tcp for application access from my personal device only
- Ingress 22/tcp for SSH access from my personal device only
- Egress ALL/tcp for ephemeral response traffic

Once deployed, Terraform presents the service's public IP address for ease of access to the endpoint.

Other features:

- t3.micro Instance Type for cost-efficiency
- SSH authorisation using PEM key for simplicity
- Hosted in default VPC for expediency
- Resources tagged with 'default tags' for cost tracking
- Terraform Makefile: consistent and concise command usage
- Go Makefile: Consistent compiling across dev environment distributions (macOS, Linux)

Possible Enhancements

The solution was devised to meet the challenge criteria in an expedient way. A real-life solution, which this is not, would take other factors into account. Here are some non-exhaustive suggestions:

The following could be implemented to improve the service's DR profile:

- Load-balance incoming requests across servers distributed across multiple Availability Zones
- Regional DR: Pilot-light or warm-standby, with requests routed via a weighted Route 53 Record
- SLIs implemented with alerting configured in-line with SLOs
- Alerting configured to scale for demand
- Runbooks devised and maintained, even automated (AWS SSM Incident Manager, AWS Lambda)
- Containerise and re-host using a Docker-compatible orchestration system (AWS ECS, AWS EKS). This will result in an exponentially faster time to reproduce the live service if an instance

becomes unhealthy. In our case, the 'scratch' image would result in re-deployments taking just a few seconds
- Use AWS Config Rules to notify when configuration drifts

The following could be implemented to improve the service's security profile:

- Activate AWS Shield Advanced on AWS ELB (if used)
- Remove Ingress 22/tcp on EC2 Security Group, in favour of administrator access via AWS Session Manager (using IAM to gain access to Terminal sessions via the AWS API)
- Implement Checkov in any CI runs
- Adopt multi-tier networking, with the application logic (and data) residing in protected subnets. Outbound requests can be routed via NAT Gateway, and for requests to the AWS API, via VPC Endpoints (AWS PrivateLink)
- Use AWS GuardDuty for threat detection, and AWS Inspector for system-level analysis (a containerised solution can use Snyk for image scanning)

The following could be implemented to improve productivity:

- Implement CI/CD pipelines to deploy infrastructure and application
- Separate infrastructure and application sources into separate repositories
- Implement linting, unit and integration testing into pipelines, including cost projection using Infracost
- If containerised and hosted in Kubernetes, adopt a GitOps workflow using tools like FluxCD or ArgoCD
- Re-produce as a serverless application, with IAC and pipelining in AWS SAM (Serverless Application Model)

Footnote

You might see in the Git commit history that the application was originally being deployed using an Ansible Playbook. This was because I had initially interpreted the brief to mean that a static application was deployed that would then be manually updated. The benefit of using Ansible here would have been that I could then effect the desired manual change from my personal device in the same way that it was deployed during bootstrap.

Once the brief was clarified however, the sections of the playbook that compiled and executed the application were transposed into the "bootstrap" Bash script, with the redundant playbook being entirely removed.