RFC

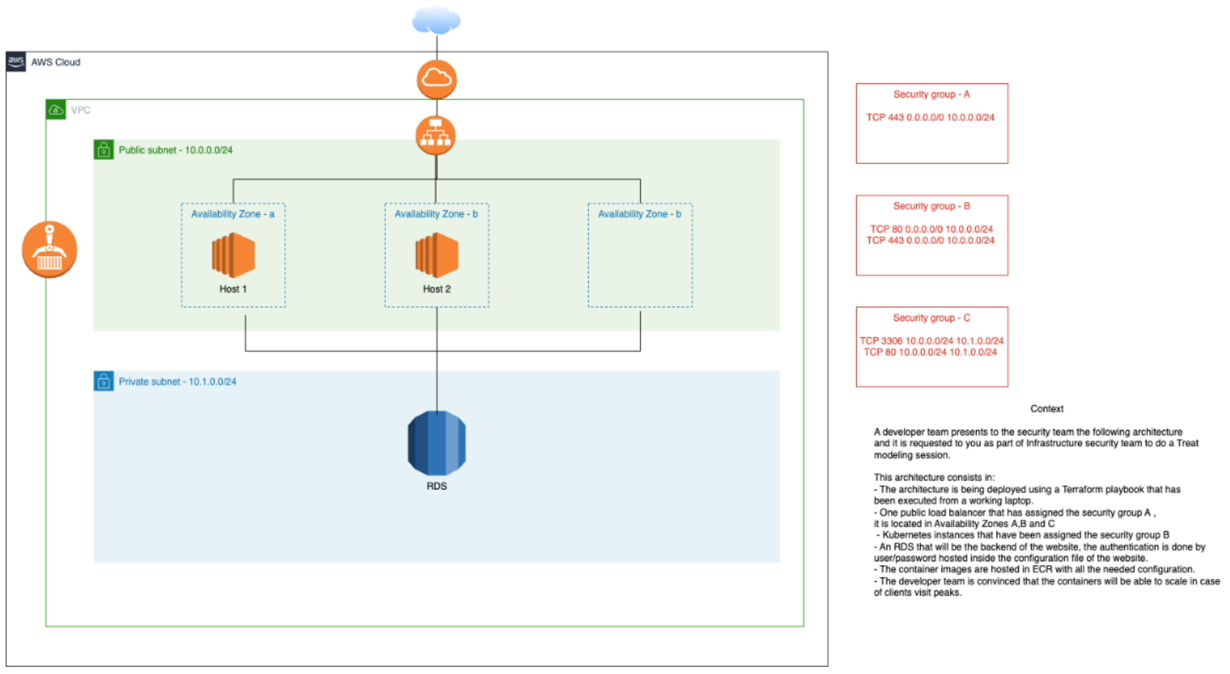
Background

Clients rely on our public services as the gateway to our offerings. Their seamless access requires uninterrupted availability, a constant assurance that every aspect of our services is accessible around the clock. Our goal is clear: to maintain continuous service, serving as the backbone of our digital presence. This steadfast commitment to availability forms the foundation of our digital domain.

Current solution

In our existing setup, we currently deploy our public web server across three EC2 instances. This configuration not only provides us with a robust and highly available infrastructure but also leverages an RDS SQL Server to enhance the efficiency and reliability of our database services. However, it's important to note that we encounter challenges during periods of high connection peaks due to the absence of an auto scalable solution. This limitation impacts our ability to seamlessly handle increased traffic demands.

Current Architecture


AWS Cloud
VPC
Public subnet - 10.0.0.0/24
Availability Zone - a
Availability Zone - b
Availability Zone - b
Host 1
Host 2
Private subnet - 10.1.0.0/24
RDS
Security group - A
TCP 443 0.0.0.0/0 10.0.0.0/24
Security group - B
TCP 80 0.0.0.0/0 10.0.0.0/24
TCP 443 0.0.0.0/0 10.0.0.0/24
Security group - C
TCP 3306 10.0.0.0/24 10.1.0.0/24
TCP 80 10.0.0.0/24 10.1.0.0/24

Context

A developer team presents to the security team the following architecture and it is requested to you as part of Infrastructure security team to do a Treat modeling session.

This architecture consists in:
- The architecture is being deployed using a Terraform playbook that has been executed from a working laptop.
- One public load balancer that has assigned the security group A , it is located in Availability Zones A,B and C
- Kubernetes instances that have been assigned the security group B
- An RDS that will be the backend of the website, the authentication is done by user/password hosted inside the configuration file of the website.
- The container images are hosted in ECR with all the needed configuration.
- The developer team is convinced that the containers will be able to scale in case of clients visit peaks.

Problems

- EC2 instances are deployed as standalone instances, this configuration does not support horizontal scaling.
- The EC2 Instances are deployed in a public subnet, this could lead to accidental exposure of internal services to the public Internet.
- The Kubernetes cluster setup lacks additional details, it is assumed that both the Kubernetes control plane and worker nodes are manually managed using EC2 compute instances.
- This configuration requires the service team to oversee patch management for both the control plane and worker nodes.
- Deployment from the developer laptop, while suitable for the dev environment, is a major security risk in production environments and also would not scale for a larger team.
- It appears credentials are stored in config files which poses security risk of credential exposure.
- Security Group rules are overly permissive.

Competing solutions

1. EC2 with Auto Scaling Group

At minimum the team could utilize EC2 Auto Scaling Group (ASG) to mitigate the scaling problem. ASG ensures the desired number of instances are always running based on the user defined scaling policies that can respond to peak demand. Additionally, the ASG instances must be created on a private subnet. The load balancer can be set up to communicate with a Target Group in a private subnet.

2. AWS Lambda Serverless

Serverless computing mitigates the operational burden associated with infrastructure (ec2 instances, k8s) management. AWS automatically replicates functions across multiple availability zones, providing built-in high availability. Nevertheless, the event-driven architectural paradigm may not align optimally with the requirements of the given application. Teams may also need to consider substantial refactoring of the application.

3. AWK Elastic Kubernetes Service (EKS)

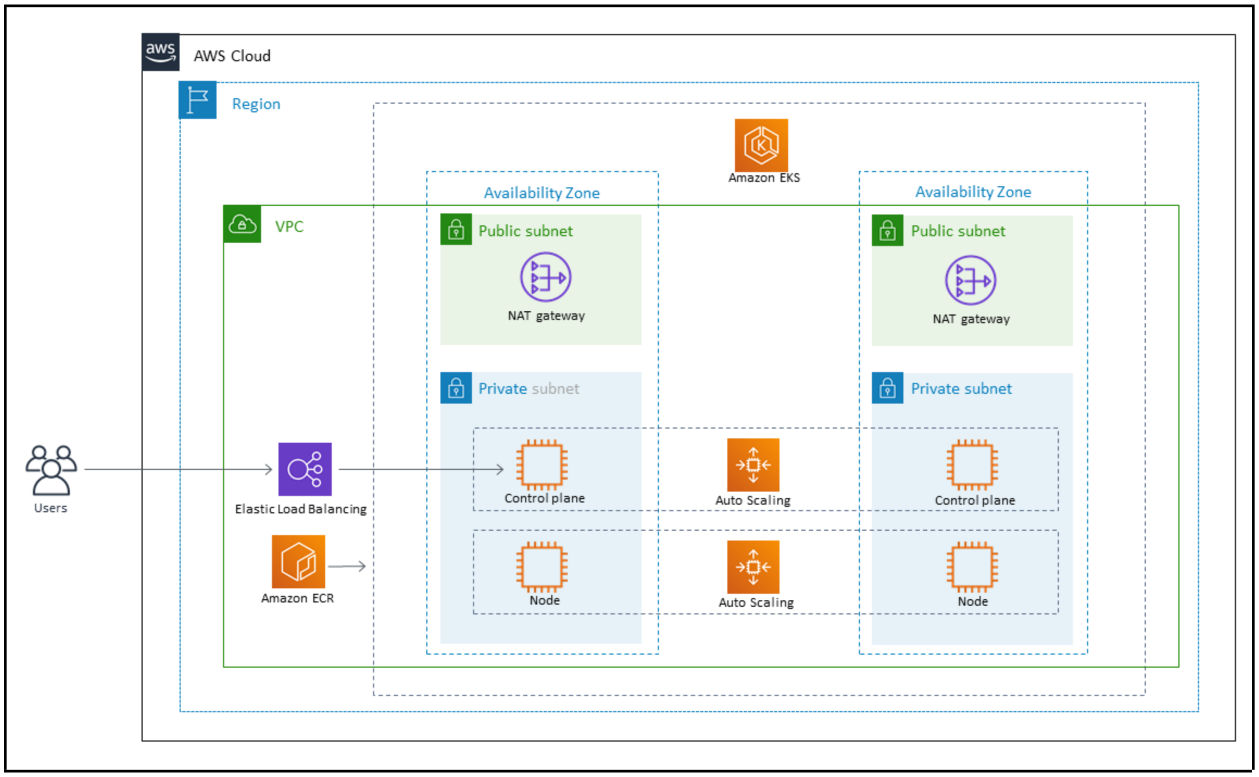
AWS EKS provides a fully managed Kubernetes control plane and an option of managed worker nodes, offloading the operational burden of maintaining and securing the components. This allows developers to focus on deploying and managing containerized applications without managing the underlying infrastructure with a higher degree of control.

Suggested Solution

The application is already using kubernetes, a managed kubernetes service like AWS EKS provides better balance of managed service and customisation with possibly least refactoring of the application architecture.

New Arch Diagram

The proposed architecture utilizes the EKS cluster for running kubernetes Control plane and Worker nodes in a private subnet. This setup minimized the attack surface while providing scalability of a managed service.



Scalability

AWS EKS supports automatic scaling, allowing for dynamic adjustment of the number of worker nodes in response to changes in demand. This ensures optimal resource utilization and responsiveness to varying workloads. Worker nodes are spread across multiple availability zones, providing built-in high availability. This ensures that applications remain resilient even if there are issues in a specific availability zone. EKS allows for the deployment of Kubernetes clusters across multiple AWS regions, providing geographic redundancy and low-latency access for users in different regions.

Dependency Consideration

- In case of self managed worker nodes, security of the worker needs to be considered.
- AWS VPC networking must be set up correctly to run worker nodes in a private subnet.
- Image factory for EC2 images should be considered for security baselined images.
- Distroless container images should be considered for minimizing attack surface.

Reliability and Security

EKS integrates with AWS IAM, allowing for fine-grained control over access to resources. Additionally, EKS benefits from AWS VPC networking, providing a secure and isolated environment for containerized workloads.

AWS is responsible for patching and updating the Kubernetes control plane in EKS. This ensures that the infrastructure remains secure and up-to-date without requiring manual intervention. Additionally, worker nodes could utilize managed Node Group to further offload the operational aspect of patching and updating.

Deployment

Implement CI/CD pipelines for IaC to automate the testing, validation, and deployment processes, ensuring that security controls are consistently applied throughout the development lifecycle. This reduces the likelihood of manual errors, enforces version control, and allows for rapid identification and remediation of security vulnerabilities in infrastructure code.

Leverage security baselined Terraform modules to promote reusability and maintainability in IaC, allowing security best practices to be encapsulated and consistently applied across different environments.

Secrets

Instead of config files, utilize environment variables for storing configuration details in IaC deployments, enhancing security by preventing sensitive information from being hardcoded in the codebase. Employing a secure secret management service such as AWS Secret Manager or HashiCorp Vault ensures the protection of sensitive data such as API keys, passwords, and other credentials.

Network

Avoid reliance on default AWS VPC configurations and public subnets in IaC deployments to enhance security by implementing a customized network architecture. Creating dedicated VPCs and private subnets provides an additional layer of isolation, reducing the attack surface and minimizing the risk of unauthorized access. By defining and managing network parameters explicitly in the IaC, we can tailor their infrastructure to meet specific security and compliance requirements.