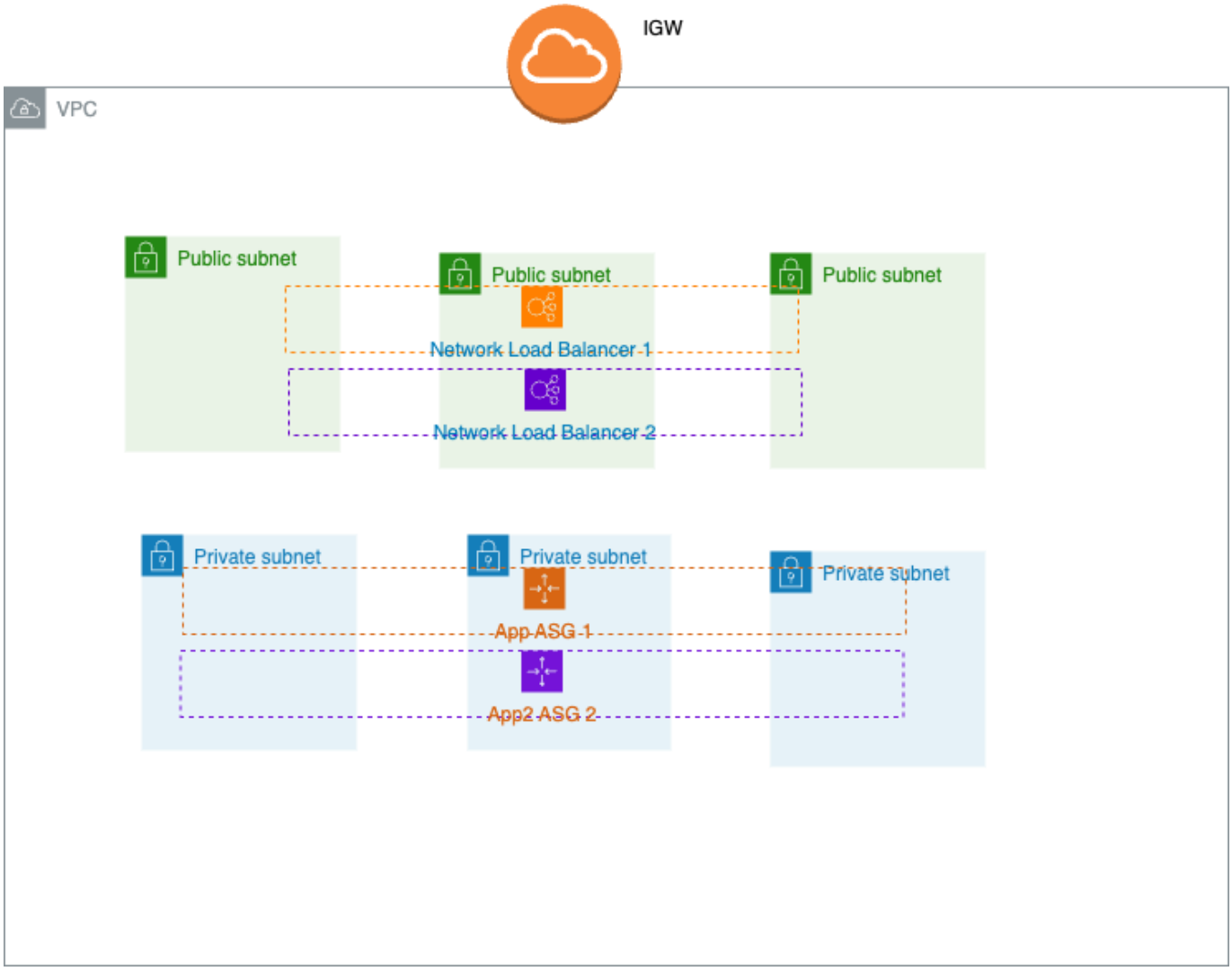
Zurich Cloud Hackathon Report

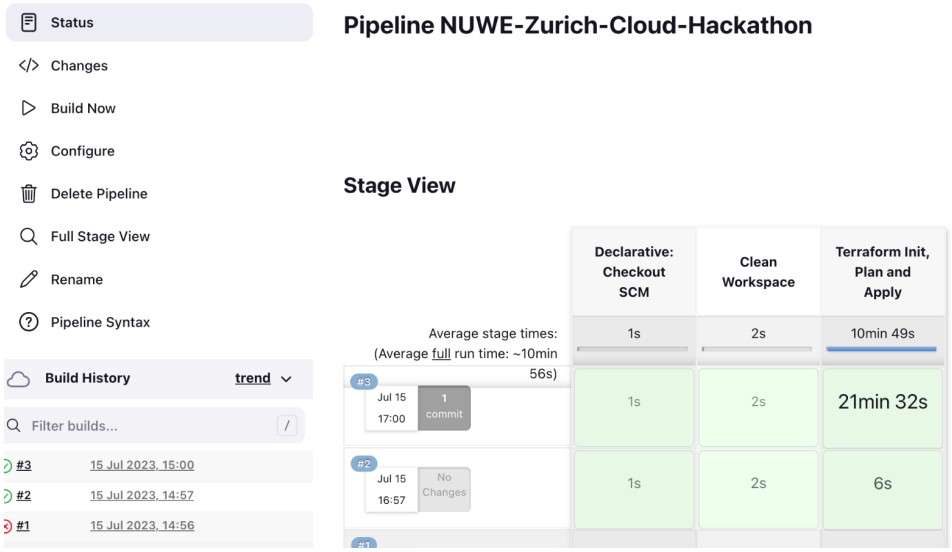
This report discusses the various architectural decisions made while designing the infrastructure for deploying services using Infrastructure as Code (IaC). Each decision is evaluated on the basis of the problem it addresses, its efficiency in comparison to other solutions, and how it optimizes costs.



The provided diagram offers a broad understanding of our infrastructure's architectural layout. It illustrates the primary components and their interactions, aiding in a comprehensive understanding of the system. Although all elements of this infrastructure are created and managed using Terraform, the diagram does not encompass all of the resources established.

The objective here is to present a clear picture of the principal structures and their relationships within the infrastructure.

Pipeline



The Jenkins file provided with the code is designed to automate the deployment of infrastructure using Terraform across multiple environments. Here is a detailed breakdown of its steps:

The Jenkins pipeline defined above is designed to automate the deployment of infrastructure using Terraform across multiple environments. Here is a detailed breakdown of its steps:

Stage 1: Clean Workspace

The pipeline starts by cleaning up the Jenkins workspace to ensure that there are no residual files from previous builds. This helps to prevent conflicts and ensures that every run starts with a clean state. It then checks out the source code from the gogs system.

Stage 2: Terraform Init, Plan and Apply

In this stage, the pipeline iterates over each environment (such as dev, staging, and production) within the "environments" directory. For each environment, the pipeline runs three key Terraform commands:

terraform init: This command initializes the working directory containing Terraform configuration files. It is safe to run this command multiple times.

terraform plan: This command creates an execution plan, determining what actions are necessary to achieve the desired state defined within the Terraform files. This provides a preview of the changes that will be made.

terraform apply --auto-approve: This command applies the desired changes to reach the desired state of the configuration.

Infrastructure Decisions and Reasons

1. Auto Scaling Groups (ASG) vs Standalone Instances

Problem Addressed

The primary requirement was to host services in two EC2 instances. However, using standalone instances doesn't offer scalability and can cause downtime during deployments.

Solution & Efficiency

Instead of deploying services on two standalone instances, the decision was made to use Auto Scaling Groups (ASG). With ASG, we can quickly scale out by increasing the number of instances when demand increases and scale in by decreasing the number of instances when demand decreases. ASG also provides rolling updates, which reduces the possibility of service interruptions during deployments.

Cost Optimization

While ASG may involve slightly less costs than standalone instances due to the maintenance of only minimum required instances, the ability to automatically scale in during periods of high demand can lead to substantial cost savings in the long run.

3. Scaling Compute based on CPU Usage

Problem Addressed

When managing a fleet of instances on AWS, there are a few challenges that developers face. One such problem is how to efficiently manage the number of running instances based on the demand or load on the servers. Not having enough instances could lead to performance degradation during peak times, while having too many would lead to cost inefficiencies.

Solution & Efficiency

The solution to the above problem is to implement Auto Scaling. Auto Scaling allows applications to adjust capacity to maintain steady, predictable performance at the lowest possible cost. This is accomplished by setting scaling policies that automatically adjust the number of active instances based on real-time load metrics.

In this particular scenario, we've used the `ASGAverageCPUUtilization` metric to control the scaling behavior. This means that the scaling is directly tied to the average CPU utilization of all the instances in the Auto Scaling group.

When the CPU usage goes above a certain threshold 75%,AWS automatically adds more instances to the Auto Scaling group ("scale out") to cope with the increased load. Conversely, when the CPU usage drops below a certain level 25%, AWS will remove instances from the Auto Scaling group ("scale in") to avoid wasteful over-provisioning.

By tracking the `ASGAverageCPUUtilization`, the system is continuously monitoring the performance of the servers and adjusting the fleet size to ensure that the CPU usage remains within the target range. This helps maintain a steady performance level and avoids the risk of overloading the servers.

Cost Optimization

Auto Scaling not only ensures that your applications are able to handle the varying load efficiently but also helps in optimizing costs.

By automatically scaling in and out based on demand, it ensures that you're only paying for what you use. When the load is high, more instances are provisioned to handle the traffic. However, when the load decreases, the extra instances are terminated. This way, you're not paying for idle resources.

Furthermore, with the option to set target CPU utilization levels, you can fine-tune the balance between performance and cost. For instance, setting a lower target CPU utilization percentage could result in fewer instances running during periods of lower demand, which could lead to cost savings. Conversely, a higher target could lead to more instances running, potentially offering better performance for your users.

However in the submitted terraform code the scaling will not happen even though it's enabled due the maximum number instances of is set to 1, but it’s easy to change the value and apply the configuration per environment.

2. Hosting ASG in Private Subnets

Problem Addressed
Network security and controlled access to instances.

Solution & Efficiency
We decided to place ASG in private subnets, significantly increasing security compared to instances directly exposed to the internet. It's assumed that either a jump host or a VPN connection is in place for SSH access, thereby adding another layer of security.

Cost Optimization
Although the private subnet's secure nature might imply slightly higher costs due to the addition of NAT gateways for outbound internet access, it ultimately saves potential costs related to handling security breaches or misconfigurations.

3. Use of Network Load Balancer (NLB)

Problem Addressed
Efficient distribution of network traffic and supporting different types of connections (TCP, UDP).

Solution & Efficiency
For routing application-related traffic, a Network Load Balancer (NLB) was used in public subnets. NLB supports both TCP and UDP traffic, providing high performance and low latency for your applications.

Cost Optimization
While NLB has some costs associated with it, it optimizes costs by effectively distributing incoming traffic, ensuring balanced utilization of resources, and reducing the need for overprovisioning.

4. Using NAT Gateways for Outbound Traffic

Problem Addressed
To enable instances in the private subnet to access the internet for updates or package downloads securely.

Solution & Efficiency
NAT Gateways situated in the public subnet were used to enable instances in the private subnet to reach the internet. This approach added a significant security layer as the private subnet's instances can access the internet without exposing their private IP addresses.

Cost Optimization
Even though this introduces additional costs, it potentially saves much larger costs associated with potential security threats and vulnerabilities that could arise from instances directly accessing the internet.

5. Encrypted S3 Buckets and IAM Roles

Problem Addressed
Secure storage of application artifacts and uploaded images controlled access to AWS services.

Solution & Efficiency
The application is zipped and stored in an encrypted S3 bucket, which provides secure, durable, and highly-scalable object storage. An IAM role is created for EC2 instances to access the S3 bucket. IAM roles provide a secure way to grant permissions to entities that you trust.

Cost Optimization
While S3 and KMS have minimal costs, they optimize costs in the long term by reducing potential security risks and associated mitigation costs.

6. Storing accidental images on S3

Problem Addressed

Storing images, particularly those related to vehicle damage, poses a challenge. The sheer volume of images can quickly exceed the capacity of traditional storage solutions, and the growth of this data is usually rapid and unpredictable. Additionally, ensuring the durability, availability, and security of this data is paramount.

Solution & Efficiency

Amazon S3 offers an ideal solution to this problem. It provides highly scalable, reliable, and secure storage for images. With its virtually unlimited storage capacity, Amazon S3 can easily handle the rapidly increasing volume of images, thereby addressing the scalability issue. The storage service is designed to deliver 99.999999999% durability, ensuring high availability and reliability for the stored images. Security is handled through powerful encryption features using KMS CMK and minimal access via AWS IAM.

Cost Optimization

One of the key considerations when managing an exponentially increasing volume of images, such as those documenting vehicle damage, is the aspect of cost. Handling large quantities of data demands an approach that not only assures availability and accessibility but also keeps the expenses under control. To address this, our chosen solution includes the advanced cost-optimization features of Amazon S3 - particularly, the S3 Intelligent-Tiering and Glacier storage classes.

Amazon S3 Intelligent-Tiering is designed for customers who have data with unknown or changing access patterns. The data is automatically moved between two access tiers -- frequent and infrequent -- based on the data's access patterns. For instance, if certain images haven't been accessed for 30 days, they are automatically moved to the infrequent access tier, reducing the cost significantly.

When the images are unlikely to be accessed for longer periods, Amazon Glacier offers an even more cost-effective solution. Glacier is Amazon's secure, durable, and extremely low-cost storage class for data archiving and long-term backup. It's designed to store data that is accessed infrequently, providing reliable storage at a fraction of the cost of other storage classes. Therefore, after a period of 60 days, we move our images from S3 Intelligent-Tiering to Glacier. This strategy drastically reduces our storage costs without sacrificing data durability or security.

Conclusion

The infrastructure was designed while considering cost efficiency, ensures scalability, high availability, and most importantly, security.