Overview

This documentation outlines the steps taken to set up and manage the infrastructure for our multi-environment (dev and prod) system. We utilized Terraform and Terragrunt to build and manage the infrastructure, leveraging both official AWS Terraform modules and custom modules to meet our specific requirements. The infrastructure includes critical components such as Terraform state management, Amazon Elastic Container Registry (ECR), Amazon Route 53, and Amazon Elastic Kubernetes Service (EKS).

Initial Infrastructure Setup

1. Terraform State Bucket

The first step in our infrastructure setup was to create a Terraform state bucket. This S3 bucket is essential for tracking the Terraform state files, ensuring that our infrastructure deployments are consistent and that state changes are properly managed across different environments. The state bucket was set up using the official AWS S3 Terraform module.

- **Why S3 for Terraform State?** S3 provides a reliable and scalable solution for storing state files. It allows for easy versioning and locking (with DynamoDB), which helps prevent concurrent operations from corrupting the state file.

2. Infrastructure Components

After setting up the Terraform state bucket, we proceeded to build the rest of the infrastructure components, which include ECR, Route 53, and EKS. The infrastructure is defined and managed using Terragrunt, with Terraform modules stored separately in the modules directory and Terragrunt configurations in the infra directory.

a. Elastic Container Registry (ECR)

ECR is used to store Docker images securely. We utilized the official AWS ECR Terraform module to create private repositories for storing our Docker images. While ECR offers an auto-scan feature for vulnerabilities, we decided to turn it off due to pricing concerns. However, we strongly recommend enabling this feature if another scanning tool is not in use. For example, we suggest using **Trivy**, an open-source vulnerability scanner, as an alternative.

- **Terragrunt Configuration:** The ECR setup is defined in the infra/dev/ecr and infra/prod/ecr directories.
- **Customizations:** The module allows for flexibility in configuring repository settings, including the option to enable or disable image scanning.

b. Route 53

Amazon Route 53 is used for DNS management. We set up Route 53 hosted zones using the official AWS Route 53 Terraform module. This setup includes managing domain names and DNS records, crucial for routing traffic to the appropriate services within our environments.

- **Terragrunt Configuration:** The Route 53 setup is located in the infra/dev/route53 and infra/prod/route53 directories.
- **Best Practices:** Ensure that DNS records are properly configured and that TTL (Time-to-Live) values are optimized for your application's needs.

c. Elastic Kubernetes Service (EKS)

EKS is the core of our infrastructure, providing a fully managed Kubernetes service. The EKS cluster was set up using the official AWS EKS Terraform module, with customizations to meet our specific needs.

- **Terragrunt Configuration:** The EKS setup is located in the infra/dev/eks and infra/prod/eks directories.
- **Custom Modules:** We utilized custom modules like kubectl manifest and helm to manage Kubernetes resources and Helm charts within the EKS cluster.

3. Multi-Environment Setup

Given the need to manage both development and production environments, we structured our infrastructure as follows:

- **Terragrunt Files:** The infra directory contains two subdirectories, dev and prod, each with its own set of Terragrunt configuration files. This structure allows for isolated and consistent management of infrastructure across environments.
- **Modules Directory:** All Terraform modules are stored in the modules directory, making it easy to reuse and maintain consistent infrastructure configurations.

Dockerfile and Security Practices

Dockerfile Creation

We created a Dockerfile to containerize our Node.js application. One key aspect of the Dockerfile is that it does not run the application as the root user. Instead, we implemented a non-root user with specific user and group IDs to enhance security.

- **Why Non-Root User?** Running applications as a non-root user mitigates the risk of privilege escalation attacks, making the container more secure.
- **Dockerfile Location:** The Dockerfile is located in the packages/service1 directory.

Docker Build and Push Pipeline

The pipeline is designed to automate the build and deployment process for the application. Upon each commit, the pipeline triggers the following steps:

1. **Build the Docker Image**: The Dockerfile is used to build the application's Docker image.
2. **Push to ECR**: The built image is automatically pushed to the corresponding ECR repository.

- **Security Scanning:** While ECR's auto-scan feature is turned off for the pricing purpose , we recommend using tools like Trivy, which is opensource, for vulnerability scanning during the build.
- **Pipeline Configuration:** The pipeline configuration is defined in the .github/workflows directory.

Kubernetes Cluster Configuration

In our infrastructure setup, we have configured a Kubernetes cluster on AWS using Amazon Elastic Kubernetes Service (EKS). This cluster is designed to be cost-effective and highly available, leveraging several best practices and AWS services to optimize both performance and security.

1. EC2 Spot Instances

Description: To manage costs effectively, our Kubernetes cluster runs on EC2 Spot Instances. Spot Instances allow us to utilize unused EC2 capacity at a significant discount compared to On-Demand Instances, making them ideal for workloads that can handle interruptions.

Key Points:

- **Cost Savings**: Spot Instances can be up to 90% cheaper than On-Demand Instances, making them an excellent choice for scalable workloads.
- **Interruption Handling**: We have configured the cluster to gracefully handle Spot Instance interruptions, ensuring minimal disruption to our applications.

Best Practices:

- **Mixed Instance Types**: We recommend using a mix of Spot and On-Demand Instances to balance cost and availability.
- **Node Auto-Scaling**: Use Kubernetes Cluster Autoscaler to automatically adjust the number of nodes in the cluster based on demand, ensuring that sufficient capacity is available even if Spot Instances are terminated.

2. High Availability with Auto-Scaling

Description: To ensure that our applications are highly available, we have implemented auto-scaling for both the nodes and pods within the Kubernetes cluster. This setup helps maintain application performance and availability, even during traffic spikes or node failures.

Key Points:

- **Node Auto-Scaling**: Automatically adjusts the number of EC2 instances in the cluster based on resource usage.
- **Horizontal Pod Autoscaling (HPA)**: Scales the number of pod replicas based on CPU and memory usage metrics, ensuring that the application can handle varying loads.

Best Practices:

- **Multiple Availability Zones**: Deploy your nodes across multiple AWS Availability Zones to increase fault tolerance.
- **Proper Resource Requests and Limits**: Set appropriate resource requests and limits for your pods to ensure the autoscaler functions effectively.

3. Kubernetes API Accessibility

Description: Currently, the Kubernetes API is externally reachable, which poses potential security risks. The best practice is to restrict access to the Kubernetes API by placing it behind a VPN, ensuring that only authorized personnel can access the cluster's management interface.

Key Points:

- **VPN Access**: Restrict Kubernetes API access to VPN users, preventing unauthorized external access.
- **Security Groups**: Use AWS Security Groups to further restrict access based on IP addresses.

Best Practices:

- **Private API Endpoint**: Enable a private API endpoint in EKS, which is only accessible within your VPC.
- **RBAC and Network Policies**: Implement Kubernetes RBAC (Role-Based Access Control) and Network Policies to control access and communication within the cluster.

4. External Service Exposure

Description: While the Kubernetes API should be secured behind a VPN, the services running inside the cluster need to be externally accessible. We achieve this using an API Gateway or Kubernetes Ingress, which acts as a reverse proxy, exposing only the necessary services to the outside world.

Key Points:

- **Ingress Controllers**: We use Kubernetes Ingress controllers to manage external access to services within the cluster. Ingress controllers provide SSL termination, load balancing, and routing based on domain names or paths.
- **API Gateway**: For more complex use cases, we utilize an AWS API Gateway to expose microservices running inside the Kubernetes cluster. This allows for fine-grained control over API routing, security, and scaling.

Best Practices:

- **Use TLS/SSL**: Always use TLS/SSL to encrypt traffic between users and your services.
- **Limit Exposure**: Only expose services that need to be publicly accessible, and ensure that other services are restricted to internal access within the VPC.

CI/CD Process

Our CI/CD (Continuous Integration/Continuous Deployment) process is designed to automate the entire pipeline from code commit to deployment on Kubernetes. The process leverages several key technologies, including ArgoCD, universal Helm charts, and SOPS encryption, to ensure that code changes are securely and efficiently deployed to our Kubernetes cluster. This document provides an overview of the tools and processes used in our CI/CD pipeline.

Key Technologies

1. ArgoCD

Description: ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes. It automates the deployment of applications to Kubernetes by monitoring a Git repository and ensuring that the desired state described in the repository matches the actual state of the cluster.

Key Features:

- **GitOps Model**: ArgoCD continuously monitors your Git repositories and automatically applies changes to the Kubernetes cluster.
- **Declarative Configuration**: Applications are defined in a declarative manner, making the deployment process predictable and reproducible.

Documentation: ArgoCD Documentation

2. Universal Helm Chart

Description: We use a universal Helm chart as a template to deploy various services to the Kubernetes cluster. Helm is a package manager for Kubernetes that simplifies the management of Kubernetes applications.

Key Features:

- **Reusable Templates**: The universal Helm chart can be reused across multiple services, reducing duplication and ensuring consistency.
- **Values Files**: Each service has specific configuration files that define its deployment parameters, including a values.yaml, version.yaml, and secrets.enc.yaml.

Documentation: Helm Documentation

3. SOPS Encryption

Description: SOPS (Secrets OPerationS) is an encryption tool used to manage sensitive information in YAML, JSON, and other file formats. We use SOPS to encrypt sensitive data, which is then stored securely in the repository and automatically decrypted by ArgoCD during deployment.

Key Features:

- **AGE Encryption**: We use AGE encryption with SOPS, which is a modern, simple, and secure encryption tool.
- **Automatic Decryption**: ArgoCD handles the decryption process automatically during the deployment phase, ensuring that sensitive information is protected throughout the CI/CD pipeline.

Documentation: SOPS Documentation

Helm Folder Structure

Within the repository, the helm folder contains the universal Helm chart value files for each service. Each service is represented by three key files:

1. **values.yaml**:
    - **Purpose**: This is the main configuration file that describes the service's deployment parameters, including resource requests, replicas, environment variables, and more.
2. **version.yaml**:
    - **Purpose**: This file contains the Docker image tag used by the service. It is automatically updated by the CI/CD pipeline after a new Docker image is built and pushed to ECR.
3. **secrets.enc.yaml**:
    - **Purpose**: This file contains sensitive information such as API keys, database credentials, and other secrets. It is encrypted using SOPS and automatically decrypted by ArgoCD during deployment.
    -

CI/CD Pipeline Process

Our CI/CD pipeline is fully automated, ensuring that when a developer pushes code to the repository, the corresponding service is automatically built, updated, and deployed to the Kubernetes cluster.

1. Build and Push Docker Image

- **Trigger**: The pipeline is triggered when code is pushed to the repository, except for changes in the helm folder.
- **Process**:
    - The pipeline builds a Docker image using the updated code.
    - The image is tagged with a unique identifier (commit SHA and timestamp) and pushed to Amazon ECR.

2. Update version.yaml

- **Process**:
    - After successfully pushing the Docker image to ECR, the pipeline updates the version.yaml file in the corresponding service's Helm folder with the new Docker image tag.
    - This update is automatically committed and pushed back to the repository.

3. Trigger ArgoCD Deployment

- **Process**:
    - The commit that updates the version.yaml file triggers ArgoCD to apply the changes to the Kubernetes cluster.
    - ArgoCD detects the updated image tag and deploys the new version of the service to the cluster.
    - If the secrets.enc.yaml file is used, ArgoCD automatically decrypts it using SOPS and applies the secrets to the cluster.

Summary

The entire CI/CD process is designed to minimize manual intervention, ensuring that code changes are automatically built, tested, and deployed to the Kubernetes cluster. Developers can focus on writing code, knowing that the deployment process is fully automated and secure, with best practices in place for handling sensitive information and managing infrastructure at scale.

Kubernetes Pod Deployment

In our Kubernetes cluster, we have implemented several best practices to ensure that our applications run securely, efficiently, and with high availability. This documentation provides an overview of how our pods are deployed and managed within the cluster, with a focus on security, resource optimization, and operational resilience.

1. Non-Root User for Docker Images

Description: To enhance security, all Docker images used in our Kubernetes cluster are configured to run as a non-root user. Running containers as non-root users minimizes the risk of privilege escalation attacks, making the environment more secure.

Key Points:

- **Security Considerations**: Running as a non-root user reduces the potential impact of a container being compromised, as the attacker would have limited privileges within the container.
- **Distroless Images**: To further reduce the attack surface and keep the Docker images as small as possible, we recommend using Google Distroless images. These images contain only the necessary runtime libraries and dependencies, with no shell or package manager, reducing the chances of vulnerabilities.

Documentation: Distroless Docker Images

2. Health Probes

Description: Each pod in the cluster is configured with health probes (liveness and readiness probes) to ensure that Kubernetes can monitor the health of the applications and manage them effectively.

Key Points:

- **Liveness Probe**: Ensures that the application is running and healthy. If the liveness probe fails, Kubernetes will restart the pod.
- **Readiness Probe**: Determines whether the pod is ready to serve traffic. If the readiness probe fails, the pod is temporarily removed from the service's load balancer until it is healthy again.

Best Practices:

- **Probe Configuration**: Properly configure the frequency and timeout of probes to balance between quick detection of failures and avoiding unnecessary restarts.

3. Resource Requests and Limits

Description: Each pod has defined resource requests and limits for CPU and memory. This ensures that the pods receive the necessary resources to operate efficiently while preventing any single pod from consuming excessive resources.

Key Points:

- **Resource Requests**: The minimum amount of CPU and memory resources that a pod needs to run efficiently.
- **Resource Limits**: The maximum amount of CPU and memory resources that a pod is allowed to use. This prevents pods from over-consuming resources, which could lead to resource starvation for other pods.

Best Practices:

- **Right-Sizing Resources**: Regularly monitor and adjust resource requests and limits to match the actual usage patterns of your applications, ensuring optimal performance and cost-efficiency.

4. Pod Affinity and Anti-Affinity

Description: We utilize pod affinity and anti-affinity rules to control the placement of pods across the nodes in the cluster. This ensures that replicas of the same application are distributed across different nodes, enhancing the availability and resilience of the application.

Key Points:

- **Pod Anti-Affinity**: Configured to ensure that no two replicas of the same application run on the same node. This prevents a single node failure from affecting multiple replicas of the same service.
- **Pod Affinity**: Can be used to ensure that certain pods are scheduled on the same node, which might be beneficial for performance reasons.

Best Practices:

- **High Availability**: Use pod anti-affinity rules to ensure high availability by spreading replicas across different failure domains (e.g., Availability Zones).

5. Rolling Updates

Description: Kubernetes is configured to perform rolling updates when deploying new versions of applications. This ensures that updates are applied without downtime, with new pods being gradually rolled out while old ones are phased out.

Key Points:

- **Zero-Downtime Deployment**: Rolling updates allow for seamless upgrades by gradually replacing old pods with new ones, ensuring that the application remains available during the update process.
- **Rollback Support**: If an issue is detected during an update, Kubernetes allows for an easy rollback to the previous version.

Best Practices:

- **Update Strategy**: Configure the rolling update strategy (e.g., maxUnavailable and maxSurge) to match your application's availability requirements during deployments.

6. Custom Service Accounts with Limited Permissions

Description: Each pod runs with a custom service account that has only the permissions necessary for the application to function. This principle of least privilege helps secure the cluster by limiting the potential impact of a compromised pod.

Key Points:

- **Service Accounts**: Custom service accounts are created for each application, with roles and role bindings that grant only the necessary permissions.
- **RBAC Policies**: Role-Based Access Control (RBAC) policies are used to enforce fine-grained access control within the cluster.

Best Practices:

- **Audit Permissions**: Regularly audit the permissions granted to service accounts to ensure they are still necessary and appropriate.

7. Horizontal Pod Autoscaling

Description: Horizontal Pod Autoscalers (HPA) are configured to automatically adjust the number of pod replicas based on resource usage metrics such as CPU and memory. This ensures that the application can scale dynamically to handle varying loads.

Key Points:

- **Resource Monitoring**: HPA continuously monitors resource usage and adds or removes replicas as needed to maintain optimal performance.
- **Scalability**: This automatic scaling helps maintain service availability during traffic spikes while optimizing resource usage during periods of low demand.

Best Practices:

- **Custom Metrics**: Consider using custom metrics (e.g., request rate, latency) in addition to CPU and memory to better capture the application's scaling needs.

8. Security Groups and Traffic Rate Limiting

Description: We use Kubernetes security groups to control network traffic to and from the pods. Security groups help in rate-limiting traffic and providing an additional layer of security by restricting access based on IP addresses and protocols.

Key Points:

- **Rate Limiting**: Security groups can be configured to limit the rate of incoming traffic, protecting the application from potential denial-of-service attacks.
- **Network Isolation**: By using security groups, we can ensure that only authorized traffic reaches the application pods.

Recommendation:

- **Istio for Advanced Traffic Management**: While security groups provide basic traffic control, we recommend using Istio, a service mesh that offers advanced traffic management, security, and observability features, including fine-grained traffic control, mutual TLS, and distributed tracing.

Custom Tools and Operators in Kubernetes

In our Kubernetes setup, we utilize several custom tools and operators to enhance the functionality and automation of our cluster. These tools ensure that our applications are securely managed, automatically updated, and seamlessly integrated with external services like DNS. Below is a list of the custom tools and operators we use, along with brief descriptions of their roles and importance in our infrastructure.

1. Secret Reloader Operator

Description: The Secret Reloader Operator is a custom Kubernetes operator that monitors changes to Kubernetes Secrets and ConfigMaps. When a secret or configmap is updated, the operator automatically restarts the associated pods to ensure that the updated information is used by the application.

Key Use Case:

- **Automatic Pod Restarts**: In Kubernetes, pods do not automatically restart when a secret or configmap is updated. The Secret Reloader Operator fills this gap by detecting changes and triggering a pod restart, ensuring that sensitive information such as API keys and passwords are always up-to-date in running applications.

Documentation: Secret Reloader on GitHub

2. ExternalDNS Operator

Description: The ExternalDNS Operator is used to automate the management of DNS records in AWS Route 53 (or other DNS providers). When a new Kubernetes Ingress resource is created, the ExternalDNS Operator automatically creates the corresponding DNS records, ensuring that the service is accessible via the specified domain name.

Key Use Case:

- **Automated DNS Management**: By automatically creating and managing DNS records, ExternalDNS removes the need for manual DNS configuration, reducing the risk of errors and ensuring that services are consistently reachable.

Documentation: ExternalDNS Documentation

3. Cert-Manager

4. Kubernetes NGINX Ingress Controller

5. Custom Helm Chart for Ingress Creation

Multitenancy in Kubernetes with Helm

Overview

In our Kubernetes infrastructure, multitenancy is implemented to manage multiple tenants efficiently. Each tenant may have unique configurations, such as environment variables or secrets, while sharing a common base service configuration. This approach ensures consistency across tenants while allowing flexibility for tenant-specific customizations. Below is a detailed description of how multitenancy is structured using Helm and symbolic links.

1. Helm Folder Structure for Multitenancy

Description: The helm folder contains a prod directory that holds the base service values for all tenants. Each tenant has its own subdirectory within the helm folder, which includes symbolic links to the shared base service values and tenant-specific configuration files.

Key Components:

- **Base Service Configuration (prod/devops-demo-base):**
    - The prod folder contains the values.yaml, version.yaml, and other essential configuration files that define the core settings for the service.

    - values.yaml: This file contains the main configuration of the service, such as resource requests, pod affinity, and rolling update strategies.

- **Tenant Directories:**
    - Each tenant has its own folder under helm, e.g., tenant-1, tenant-2.
    - Symbolic Links: The values.yaml file in each tenant directory are symlinked to the base configuration in the prod folder. This ensures that any updates to the base configuration or version affect all tenants uniformly.
    - env.yaml: A tenant-specific environment file that defines environment variables unique to that tenant. This file is not symlinked, allowing each tenant to have custom environment settings.
    - secrets.enc.yaml: Contains tenant-specific secrets, encrypted using SOPS. These files are unique to each tenant and are not shared or symlinked.

Benefits:

- Consistency: By symlinking the base values.yaml file, we ensure that any updates to the service configuration or Docker image tag are automatically propagated to all tenants.
- Flexibility: Tenant-specific configurations (like environment variables and secrets) are easily managed without affecting other tenants, allowing for tailored configurations per tenant.

2. Managing Updates

- **Base Service Updates:**
    - When the values.yaml or version.yaml files in the prod directory are updated, all tenants automatically receive these updates because their symlinked files point to the base service files. This ensures that all tenants are consistently using the latest configurations and Docker images.
- **Tenant-Specific Updates:**
    - When only the env.yaml or secrets.enc.yaml file in a tenant directory is updated, the changes apply only to that specific tenant. The tenant's service is updated without impacting the configurations or operations of other tenants

URL's for the endpoints:

Dev: devops-demo.chene-factory.com

Prod/tenant1: devops-demo-tenant-1.chene-factory.com

Prod/tenant2: devops-demo-tenant-2.chene-factory.com