Technical Document for Nuvolar API Service

1. Introduction

This document outlines the tools, methodology, and Continuous Integration/Continuous Deployment (CI/CD) strategy for Nuvolar API service. The API consists of six microservices that communicate via RabbitMQ, with some microservices using a RDS Cluster for data persistence. The system is designed to handle high concurrency. It's deployed in four environments: develop, test, stage and production.

2. System Architecture

The system architecture comprises the following key components:

- **Six Java Microservices:** Each microservice is a self-contained application.
- **Docker:** Microservices are containerized using Docker.
- **RabbitMQ:** A message broker used for asynchronous communication between microservices.
- **RDS Cluster:** Used by stateful microservices.
- **HTTP API:** The application is accessed via an HTTP API, using Load Balancer

3. Tools and Technologies

The following tools and technologies are selected for the CI/CD pipeline and infrastructure:

- **Version Control:**
  - **GitHub:** A Git repository hosting service, mono-repos are used for all services.
- **Build Tool:**
  - **Maven:** A build automation tool used for Java projects**.**
- **Containerization:**
  - **Docker:** Used for dockerization of the services.
- **Message Broker:**
  - **RabbitMQ:** An open-source message-broker software that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), MQ Telemetry Transport (MQTT), and other protocols.
- **Database:**

    - **RDS Cluster:** EKS managed DB instance
- **Orchestration:**
    - **EKS:** AWS managed Kubernetes A container orchestration platform.
        - *Reasoning:* Automated deployment, scaling, and management of the Dockerized microservices. Provides features like service discovery, load balancing, and self-healing, crucial for a high-concurrency, microservices-based system.
- **CI/CD Server:**
    - **GitHub Actions for CI:** GitHub is used as a Git server, and the integrated pipelines are preferred for CI/CD especially for mono-repos.
    - **ArgoCD for GitOps CD**: Helm files are hosted on a dedicated repository.
- **Infrastructure as Code (IaC):**
    - **Terraform:** Infrastructure as code software tool created by HashiCorp.
- **Monitoring and Logging:**
    - **Prometheus:** For metrics collection and monitoring.
    - **Grafana:** For visualizing metrics from Prometheus.
    - **Elasticsearch, FluentBit and Kibana (EFK stack):** For centralized logging.
- **API Gateway (Optional):**
    - **Kong, Nginx, or Traefik:** To manage external API requests.
        - *Reasoning:* Can provide routing, authentication, and other functionalities.

4. Development Workflow

The development workflow follows a Git-based approach:

1. **Feature Branching:** Developers create feature branches from the main branch for each new feature or bug fix.
2. **Code Development:** Developers implement the changes in their feature branches.
3. **Code Review:** Before merging, code undergoes review by other developers using pull requests, also there are code quality checkers
4. **Merge to develop branch:** Once approved, the feature branch is merged into the develop branch.
5. **Propagation to "higher" branches(test,stage,main):** same strategy, the flow is develop -> test -> stage -> main, merge is possible only using pull request

5. CI/CD Pipeline

The CI/CD pipeline automates the process of building, testing, and deploying the microservices. A multi-branch pipeline is recommended.

5.1 Continuous Integration (CI)

The CI pipeline is triggered automatically on every commit to the main branch (and potentially on Pull Requests).

- **1. Code Checkout:** The pipeline retrieves the latest code from the Git repository.
- **2. Static Code Analysis:** SonarQube is integrated to check for code quality, security vulnerabilities, and coding standards compliance.
- **3. Compilation:** Maven pulls dependency and compiles the Java code.
- **4. Unit Testing:** Maven executes unit tests.
- **5. Docker Image Build:** A Docker image is created for the microservice.
- **6. Docker Image Push:** The Docker image is pushed to a Docker registry (AWS ECR registry)
- **7. Updating Helm files:** The CI pipeline updates Helm files hosted on a dedicated repository used by ArgoCD with a new Image tag. Every environment has different values.yaml file which define config values for that environment.

5.2 Continuous Deployment (CD)

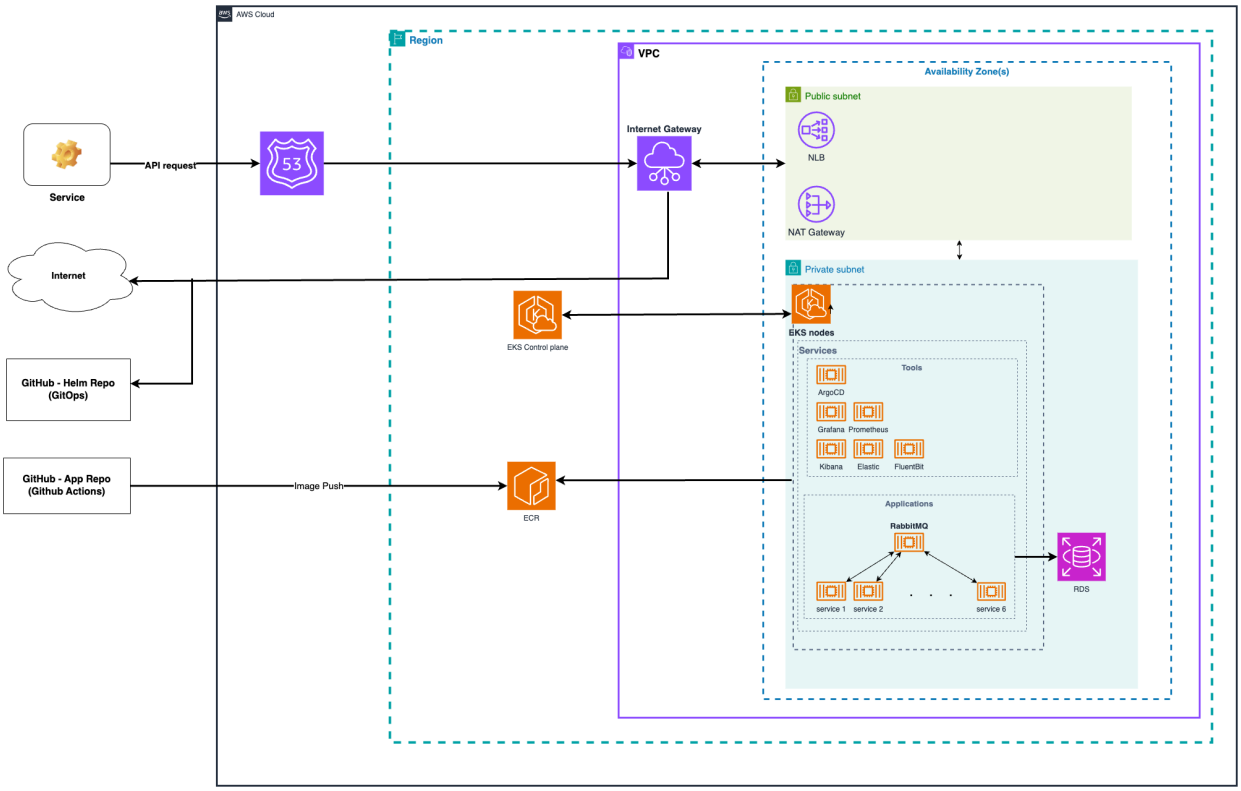
The CD pipeline automates the deployment of new microservice versions to a target environment (develop, test, staging, production).

- **Deployment to Develop, Test and Stage environment:**
  - **1. ArgoCD check:** The ArgoCD server checks for changes made on Helm chart repository.
  - **2. Helm Configuration Update:** If the configuration file is updated, ArgoCD will automatically trigger a Kubernetes rolling update. The environment where ArgoCD deploys the new version is determined by the modified values files.
  - **3. Notification:** After a successful or failed update, ArgoCD will send a notification to the designated Slack channel.
- **Deployment to Production:**
  - **Same steps apply as for "lower" environments;** the only difference is that for production, ArgoCD prepares the deployment, but the **sync is performed manually.**

6. Infrastructure Provisioning

- **Infrastructure as Code (IaC):** Terraform is used to define and provision the infrastructure required to run the application. This includes:
  - EKS deployment, which includes:
    - Configure compute nodes
    - Configure addons and operators
    - Configure monitoring and logging infrastructure

    - RDS DB deployment.
    - Load balancer and DNS configuration.
- **Infrastructure diagram:**



7. Concurrency setup

- **Horizontal Scaling:** Kubernetes is used to scale microservices horizontally by increasing the number of pods, metric used: memory usage, cpu usage and number of requests.
- **Managed Node Groups Autoscaler:** Automatically increases the number of EC2 instances in the node group to accommodate pending pods that cannot be scheduled due to insufficient resources.
- **Load Balancing:** Kubernetes Service provides load balancing across microservice instances. A Layer 7 load balancer or API Gateway can provide more advanced routing and traffic management.
- **Database Scalability:** RDS DB is designed to handle high concurrency by distributing data and load across multiple nodes.
- **Message Queue Scalability:** RabbitMQ can be clustered to handle a high volume of messages.

8. Monitoring and Logging

- **Metrics Monitoring:**
  - Prometheus collects metrics from the microservices (e.g., CPU usage, memory usage,number of requests, request latency, error rates).
  - Grafana visualizes these metrics in dashboards, providing insights into system performance and health. Alerts are configured to notify operators of critical issues.
- **Centralized Logging:**
  - FluentBit collects logs from the microservices and sends them to Elasticsearch.
  - Elasticsearch stores and indexes the logs, making them searchable.
  - Kibana provides a web interface for searching, analyzing, and visualizing the logs.

9. Desirable improvements

- Set up an API Gateway for better API management.
- Configure AWS WAF (Web Application Firewall) rules
- Improve Secrets Management by using a secure solution like HashiCorp Vault or AWS Secrets Manager to store secrets (currently, secrets are stored in Kubernetes Secrets which are not secure).
- Implement Docker security/vulnerability scanners like Trivy
- Implement Karpenter for efficient autoscaling and resource management. Karpenter automatically provisions and scales your Kubernetes nodes based on resource demand, improving cluster efficiency.
- Set up Network Policy Engines such as Calico or Cilium for better control over service-to-service communication in Kubernetes.
- Implement a Service Mesh (e.g., Linkerd or Istio) for traffic management, observability, and security.

.