

End-to-End DevOps Project: 3-Tier Microservice E-Commerce Application with Continuous Integration, Deployment, and Monitoring in AWS

Introduction:

In this project, we embark on an ambitious journey to deploy and monitor a 3-tier microservice web application using a comprehensive DevOps approach on AWS. Leveraging the "Stan's Roboshop" project as our case study, we aim to implement an end-to-end DevOps pipeline that ensures robust deployment, scalability, and continuous monitoring of the application.

Project Overview:

Stan's Roboshop is a modern e-commerce platform built using a microservices architecture. It consists of multiple interconnected services, each responsible for different aspects of the application, such as the user interface, business logic, and data storage. The primary goal of this project is to deploy these services efficiently based on developer requirements using best-in-class DevOps practices and tools.

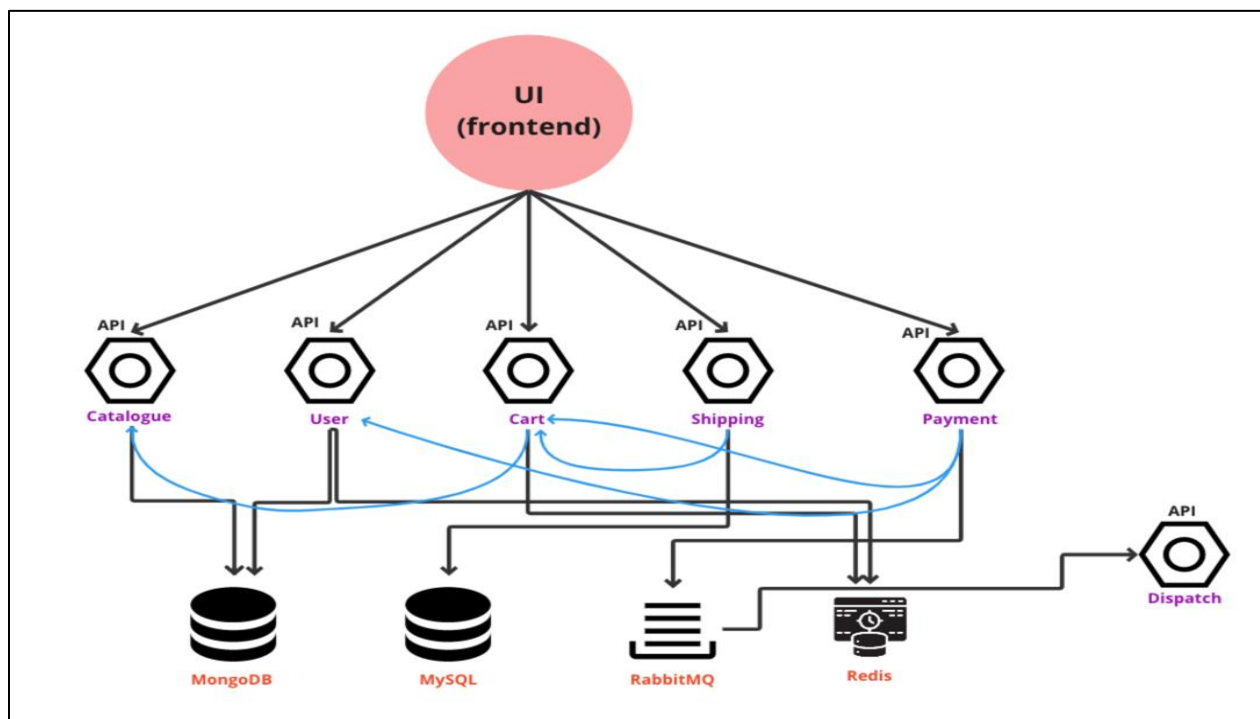


Figure 1

1. Frontend:

The frontend service in RoboShop is responsible for serving the web content using Nginx. This includes the static content and web framework for the application. A web server is required to serve the static content effectively. The developer has chosen Nginx as the web server for this purpose. Therefore, we will install and configure the Nginx web server to handle the frontend service.

2. MongoDB:

The developer has selected MongoDB as the database for RoboShop. We will install and configure MongoDB to handle the application's data storage needs. This involves setting up MongoDB on the server, configuring the necessary settings, and ensuring it integrates seamlessly with other services in the application.

3. Catalogue:

The Catalogue microservice in RoboShop is responsible for serving the list of items displayed in the application. It handles item data retrieval and presentation, ensuring that users can view available products. We will deploy and configure this microservice to integrate with the other components of the RoboShop application.

4. Redis:

Redis is utilized for in-memory data storage and caching in the RoboShop application. It enhances performance by allowing quick access to frequently accessed data. Redis interacts with the database and provides data to users via APIs, ensuring efficient and fast data retrieval for the application.

5. User:

The User microservice in RoboShop handles user logins and registrations. It manages user authentication, ensuring secure access to the e-commerce portal. This service is crucial for maintaining user accounts and providing a seamless login and registration experience for RoboShop customers.

6. Cart:

The Cart microservice in RoboShop manages the shopping cart functionality. It handles adding, removing, and updating items in the cart, ensuring a smooth shopping experience for users. This service is essential for tracking user-selected products and preparing them for the checkout process.

7. MySQL:

The developer has chosen MySQL as the database for shipping services in RoboShop. We will install and configure MySQL to manage and store relational data. This setup involves database creation, user management, and ensuring proper integration with the application's microservices to facilitate efficient data handling.

8. Shipping:

The Shipping service in RoboShop is responsible for calculating the shipping distance and price for packages. This service is written in Java, so we need to install Java. Additionally, Maven, a Java packaging tool, will be installed to manage dependencies and build the Java application. Maven installation will also ensure that Java is installed.

9. RabbitMQ:

RabbitMQ is a messaging queue used by payment services in the RoboShop application. It facilitates asynchronous communication between microservices, ensuring reliable message delivery and scalability. RabbitMQ's robust messaging capabilities enable efficient handling of tasks and events across different parts of the application architecture.

10. Payment:

The Payment service in RoboShop handles all payment transactions within the e-commerce application. Developed using Python 3.6, it ensures secure and reliable processing of payments from customers. This service integrates with payment gateways to provide a seamless checkout experience, ensuring transactions are handled efficiently and securely.

11. Dispatch:

Dispatch is the service responsible for product dispatch after purchase in the RoboShop application. Developed in GoLang, it efficiently manages order fulfilment and ensures timely delivery to customers. This service integrates with logistics systems to track and manage the shipment process, ensuring smooth operations for order dispatching.

Objective:

In this DevOps project, we aim to achieve several key objectives to ensure efficient development, deployment, and operation of the application. These objectives span across automation, containerization, orchestration, configuration management, code quality, artifact management, and application monitoring.

1. Automated Deployment:

Our first objective is to implement Continuous Integration/ Continuous Deployment (CI/CD) pipelines using Jenkins. This involves automating the building, testing, and deployment processes of the microservices that constitute our 3-tier web application. Jenkins will orchestrate the entire pipeline, ensuring that new code changes are automatically built, tested, and deployed to production or staging environments. This automation reduces manual effort, accelerates time-to-market, and enhances the overall efficiency of our development lifecycle.

2. Containerization:

Using Docker, we will containerize each microservice of our application. Containerization ensures that each service and its dependencies are encapsulated into lightweight, isolated containers. This approach enhances consistency across different environments, whether it's development, testing, or production and facilitates seamless deployment and scaling of microservices. Docker containers provide portability, allowing us to deploy applications quickly and efficiently across various platforms and environments.

3. Orchestration:

Kubernetes will play a pivotal role in orchestrating our containerized microservices. Deployed on an AWS Kubernetes Service (EKS) cluster, Kubernetes uses Helm charts to define and manage applications. This orchestration framework ensures high availability, scalability, and efficient resource utilization across our microservices. Kubernetes automates deployment, scaling, and management tasks, thereby, enhancing the reliability and performance of our application infrastructure on AWS.

4. Configuration Management:

To automate the provisioning and configuration of both infrastructure and application deployments, we will employ Ansible. Ansible playbooks will define the desired state of servers and applications, ensuring consistency and reproducibility across different environments. This approach minimizes configuration drift, streamlines management tasks, and enhances system stability and security by enforcing consistent configurations throughout the deployment lifecycle.

5. Infrastructure Management:

Terraform will serve as our tool of choice for managing Infrastructure as Code (IaC) on AWS. By defining infrastructure resources such as virtual machines, networks, and storage in a declarative manner, Terraform ensures consistency and scalability of our underlying infrastructure. This approach enables us to provision and manage AWS resources efficiently, supporting the needs of our microservices and Kubernetes cluster with agility and reliability.

6. Code Quality:

Integrating SonarQube into our CI/CD pipeline will enable continuous static code analysis. SonarQube will scan our codebase for code smells, bugs, vulnerabilities, and maintainability issues. By enforcing coding standards and best practices, SonarQube enhances overall code quality, security, and maintainability of our application. This proactive approach ensures that potential issues are identified early in the development process, promoting a robust and reliable codebase.

7. Artifact Management:

Nexus Repository will act as the central repository for storing and managing build artifacts generated throughout our CI/CD pipeline. Nexus ensures efficient artifact distribution, version control, and collaboration among development teams. By providing reliable artifact storage and management capabilities, Nexus supports seamless deployment processes and enhances the efficiency of our development and release cycles.

8. Application Monitoring:

To ensure comprehensive monitoring of our deployed application, we will implement various tools:

1. Prometheus and Grafana:

Prometheus will collect metrics from Kubernetes pods and microservices, offering insights into performance and resource utilization. Grafana will visualize these metrics through customizable dashboards, enabling proactive monitoring, analysis, and troubleshooting of issues.

2. ELK Stack (Elasticsearch, Logstash, Kibana)

Elasticsearch will centralize logs across our microservices, providing a unified platform for log storage and retrieval. Logstash will process and forward logs to Elasticsearch, ensuring that logs are structured and easily searchable. Kibana will offer a user-friendly interface for log visualization, analysis, and real-time monitoring across our application stack.

3. New Relic:

New Relic will monitor application performance in real-time, offering insights into application behaviour, transaction traces, and infrastructure metrics. Its comprehensive monitoring capabilities will help identify performance bottlenecks, optimise application performance, and ensure a positive user experience. Here's a summary below of the tools and technologies.

Tools and Technologies:

- **AWS (Amazon Web Services):** Cloud platform for hosting the infrastructure
- **Jenkins:** CI/CD tool for automating the build and deployment pipelines
- **SonarQube:** Platform for continuous inspection of code quality
- **Nexus Repository:** Artifact repository manager

- **Git/ GitHub:** Source code management (SCM) and version control system
- **Ansible:** Configuration management tool for automating setup and deployment tasks
- **Terraform:** Infrastructure as Code (IaC) tool for provisioning and managing AWS resources
- **Docker:** Containerization platform for packaging microservices
- **Kubernetes:** Container orchestration platform for managing containerized applications
- **Helm:** Kubernetes package manager for deploying applications.
- **ELK Stack (Elasticsearch, Logstash, Kibana):** Centralized logging solution.
- **Prometheus & Grafana:** Monitoring tools for metrics collection and visualization.
- **New Relic:** Application performance monitoring tool.
- **Shell and Python scripting:** For automating and scheduling jobs.

Scope and Implementation:

The project implementation involves setting up an AWS environment to host our Kubernetes cluster and other necessary services. Using Terraform, we will provision and manage our AWS resources, ensuring a scalable and reproducible infrastructure setup.

- We will create Jenkins pipelines for continuous integration and deployment, ensuring each microservice is built, tested, and deployed seamlessly. Docker (ECR) will be used to containerize the services, and Kubernetes (EKS), managed via Helm, will handle orchestration. Ansible will automate the configuration and deployment tasks, ensuring consistency and repeatability.
- For monitoring purposes, Prometheus and Grafana will be deployed to collect and visualize metrics, while the ELK stack will handle logging. New Relic will provide insights into application performance, helping us identify and resolve issues promptly.
- By the end of this project, we will have a fully automated, scalable, and monitored 3-tier microservice web application deployed on AWS, demonstrating the power and efficiency of modern DevOps practices.

Types of environments for IT infrastructure, deployment, and monitoring:

1. **On-Premise**
2. **Cloud**
3. **Hybrid**

1.1 On-Premise: On-premise IT infrastructure involves hosting all hardware and software within a company's physical location. This setup offers full control, security, and compliance but requires significant capital and maintenance.

1.2 Cloud: Cloud environments utilize external providers like AWS, offering scalable, flexible, and cost-effective resources. They reduce the need for physical hardware and support rapid deployment.

1.3 Hybrid: Hybrid environments combine on-premise and cloud solutions, leveraging the control and security of on-premise with the scalability and flexibility of the cloud, providing a balanced approach to resource management.

Note: In our DevOps project, we are using the AWS cloud and Services.

AWS (Amazon Web Services):

Amazon Web Services (AWS) stands as a premier cloud computing provider renowned for its extensive array of services. From foundational computing resources like EC2 instances to scalable storage solutions such as S3 and comprehensive managed database options like RDS, AWS empowers businesses with flexible, secure, and cost-effective cloud infrastructure. With a global presence and robust security measures, AWS enables organisations to innovate rapidly, streamline operations, and achieve unparalleled scalability and reliability in their digital endeavours.

1. Setting up network for Dev and Prod environments:

In traditional network setups, the choice of IP range often revolved around accommodating server needs, leading to potential migrations for additional capacity. To mitigate complexity, larger IP ranges became common practice, allowing flexibility in resource utilization. Presently, designing VPCs with a /16 network range for both development (10.0.0.0/16) and production (10.20.0.0/16) environments aligns with AWS's support for expansive scalability, accommodating up to 65,534 IPs per network. This approach optimises infrastructure management by providing ample room for growth and simplifying network administration across distinct developmental stages and production deployments.

***Development (Dev)** environment, we utilize **10.0.0.0/16** network range.

***Production (Prod)** environment, we utilize **10.20.0.0/16** network range.

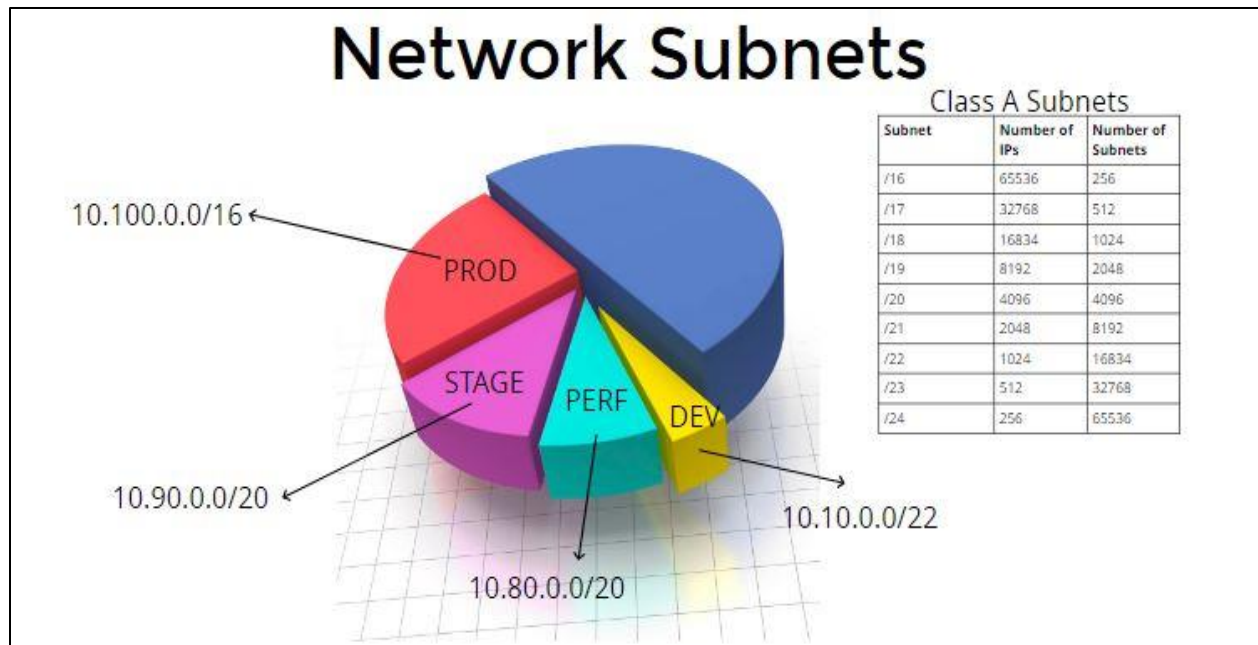


Figure 2

2. VPC (Virtual Private Cloud):

A Virtual Private Cloud (VPC) in AWS is a foundational component that enables you to create a logically isolated virtual network within the AWS cloud environment. It allows you to define your own IP address range, create subnets, configure route tables, and gateways, providing complete control over your network architecture and resources.

VPCs enhance security by allowing you to set up network access control lists (ACLs) and security groups to regulate inbound and outbound traffic. This isolation ensures that resources within different VPCs remain securely separated unless explicitly configured otherwise.

Furthermore, VPCs support connectivity options that enable seamless integration with other AWS services, such as Amazon EC2 instances, Amazon RDS databases, and AWS Lambda functions. They also facilitate connections to on-premises data centres via AWS Direct Connect or VPN, enabling hybrid cloud deployments.

With features like Elastic Load Balancing for distributing incoming application traffic, AWS Private Link for securely accessing AWS services without exposing data to the public internet, and VPC Peering for connecting multiple VPCs, AWS VPCs provide the flexibility, scalability, and security required for building sophisticated cloud architectures tailored to diverse business needs.

3. VPN (Virtual Private Network):

A Virtual Private Network (VPN) establishes a secure, encrypted connection over a less secure network, such as the internet, enabling users to securely access private networks from remote locations. VPNs ensure confidentiality by encrypting data in transit, protecting it from interception or eavesdropping. They also authenticate users to ensure only authorized individuals gain access. VPNs are crucial for remote workforces needing secure access to corporate resources, as well as for individuals seeking to browse the internet privately. Technologies like SSL VPNs and IPsec VPNs provide different methods for secure connectivity, offering flexibility in deployment based on organizational needs for privacy, security, and accessibility.

In enterprises, AWS Direct Connect is preferred for high-bandwidth, low-latency connections, whereas smaller organisations often utilise AWS VPN for secure connectivity. For our project, we are employing a workstation EC2 instance as a bastion host. This setup allows us to securely connect to our VPC using SSH, effectively simulating VPN-like access without the need for additional VPN software, thus providing a cost-effective and secure solution for our lab environment.

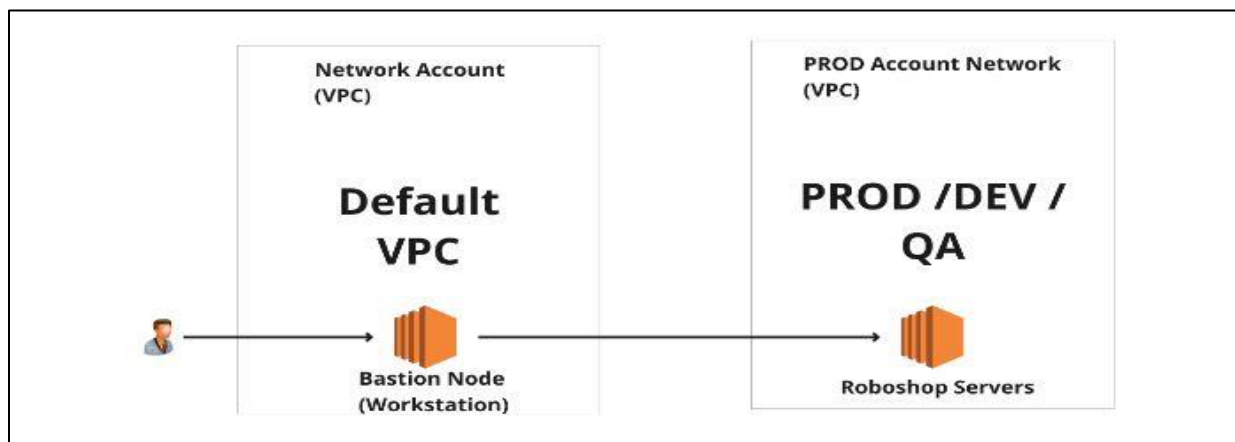


Figure 3

4. Bastion node:

A bastion node, bastion host, or jump server is a secure server used to provide controlled access to a private network from an external network. It acts as a gateway, allowing users to connect via SSH or other protocols, enhancing security by limiting exposure of internal resources. The bastion host is typically hardened and monitored, serving as a single point of entry to the network, which simplifies security management and minimises the attack surface for potential intruders.

Note: We will use our "Workstation" as a bastion host, which is located in the default VPC. We will connect to the subnets in the Dev-VPC and Prod-VPC using VPC Peering, but we will not utilize the default VPC for our VPC and VPC Peering connections.

5. VPC Peering:

VPC peering in AWS allows secure communication between Virtual Private Clouds (VPCs) within the same or different AWS accounts. It enables instances in one VPC to directly communicate with instances in another VPC using private IP addresses. VPC peering does not involve gateway devices or require internet access, making it ideal for connecting VPCs across different regions or accounts. It simplifies network management and facilitates resource sharing while maintaining isolation and security between VPCs.

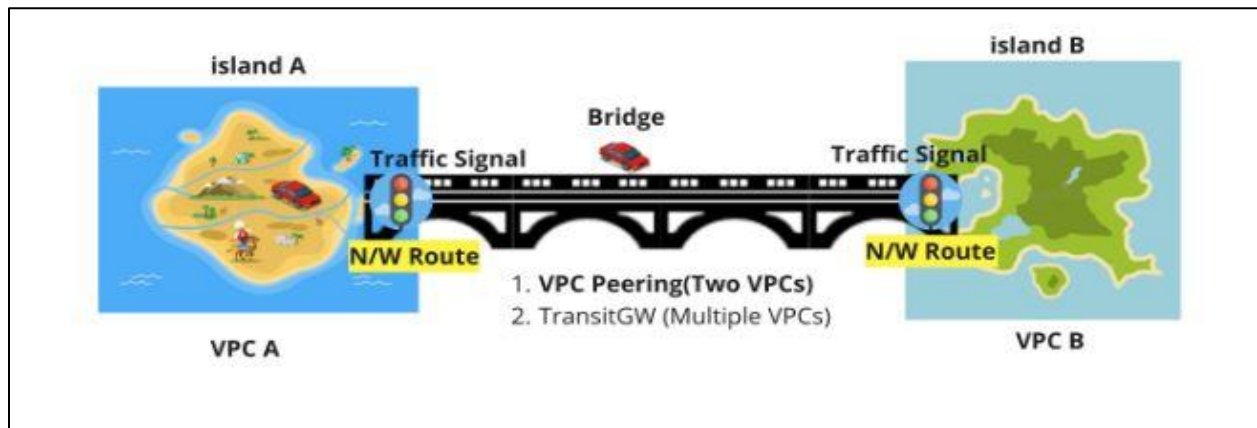


Figure 4

1. We establish connectivity between two VPCs using VPC peering.
2. To initiate VPC peering, ensure there are no network conflicts.
3. VPC peering supports connections:
 - within the same AWS account and regions.
 - across the same AWS account but different regions.
 - between different AWS accounts spanning the same or different regions.

Note: Our DevOps project utilizes VPC peering within the same AWS account and region.

6. RTB (Route Tables):

A route table in AWS defines how network traffic is directed within a Virtual Private Cloud (VPC). It contains rules or routes that determine where specific types of traffic should go. For communication between two VPCs, each VPC must have a route entry pointing to the VPC peering connection or VPN gateway. Without these route entries correctly configured in the route tables of both VPCs, network traffic cannot be properly directed, resulting in failed communication between the VPCs.

We configure route tables in the default VPC, Dev-VPC, and Prod-VPC to facilitate VPC peering connections and manage traffic routing effectively. This setup enables SSH access to instances using private IPs. By updating network routes in the route tables, we allow specific traffic to

traverse both sides, ensuring secure and efficient communication across interconnected VPCs via VPC peering. Route tables are necessary to specify how traffic flows between VPCs; without them, connections fail due to routing misconfigurations.

Note: In our DevOps project, we are creating 4 (Four) route tables.

- Public RTB
- Web RTB
- App RTB
- DB RTB

7. Internet Connection to Servers:

Servers in AWS Virtual Private Clouds (VPCs) access the internet through public subnets utilizing internet gateways (IGWs) for outbound routing. Instances in public subnets can have public IP addresses assigned directly or through Elastic IPs (EIPs) for persistent connectivity. For instances in private subnets, outbound internet access is facilitated via Network Address Translation (NAT) gateways or instances. Security groups and network ACLs enforce access controls, regulating inbound and outbound traffic flow. This configuration ensures servers can securely communicate externally while maintaining network isolation within the VPC, adhering to AWS best practices for scalable and secure cloud deployments.



Figure 5

- Subnets attached with **Internet Gateway** are called as Public Subnets.
- Subnets attached with **NAT Gateway** are called as Private Subnets.

8. IGW (Internet Gateway):

An Internet Gateway (IGW) in AWS facilitates bidirectional communication between instances in a Virtual Private Cloud (VPC) and the internet. It serves as a scalable and highly available gateway for outbound traffic from instances in public subnets, enabling them to access and interact with external services and resources securely. IGWs play a crucial role in providing connectivity for applications that require reliable and efficient access to internet-based services while maintaining network isolation and security within the AWS cloud infrastructure.

We are adding Internet Gateway (IGW) only to dev-public and prod-public avoiding dev-web, prod-web, dev-app, prod-app, dev-db, prod-db.

9. NGW (Nat Gateway):

Network Address Translation (NAT) in AWS enables instances in private subnets to initiate outbound communication to the internet or other AWS services while remaining inaccessible from the internet. NAT gateways or instances translate private IP addresses to public IPs, allowing instances to access external resources securely. This setup enhances security by hiding internal IP addresses and simplifies network management by centralizing internet connectivity through a single gateway, ensuring controlled and secure outbound traffic flow from private subnets.

Note: Creating a NAT Gateway in AWS necessitates the use of an Elastic IP (EIP). This EIP acts as a static, public-facing IP address that serves as the gateway for outbound internet traffic from instances in private subnets. It provides a reliable point of reference for external internet access, ensuring secure connectivity for all servers within the subnet without exposing their private IP addresses.

Note: In our DevOps project, we are attaching NAT Gateways to dev-web, prod-web, dev-app, prod-app, dev-db, prod-db.

Public Subnets (dev-public, prod-public) attach IGW

Private Subnets (dev-web, prod-web, dev-app, prod-app, dev-db, prod-db) attach NGW

10. E IP (Elastic IP):

An Elastic IP (EIP) in AWS is a static IPv4 address designed for dynamic cloud computing scenarios. It allows AWS instances to maintain a consistent public IP address across reboots and instance replacements. EIPs are essential for applications requiring predictable public-facing endpoints, enabling seamless failover, load balancing, and reliable access without the need for frequent IP address changes or DNS updates.

11. Route Table association with Subnets:

We associate subnets with route tables to define distinct routing paths. Public subnets link to route tables with routes enabling internet access, while private subnets connect to route tables for internal VPC communication. This setup ensures that instances in public subnets access the internet via specified routes, while those in private subnets communicate securely within the VPC. Thus, we restrict internet access to private subnets and their servers solely through the “Workstation” server, enhancing security and control over network traffic.

Note: The default route table is not adding peering connection through our terraform code.

12. Subnets:

Subnets in AWS VPCs are logical segments of IP addresses within the VPC's CIDR block. They are used to organize and manage network resources based on security and operational needs. Subnets can be public, allowing instances to have direct internet access via an Internet Gateway (IGW), or private, using Network Address Translation (NAT) gateways or instances for outbound connectivity. Subnets provide flexibility in deploying and isolating resources while adhering to security and operational requirements within the VPC environment. We are having two subnets in our VPC:

1. Public-Subnet (Public Load Balancer (ALB))

2. Private-Subnet (Web-Subnet, APP-Subnet, DB-Subnet, Private Load Balancer (ALB))

- Web-Subnet (Frontend (ASG))
- APP-Subnet (Cart (ASG), Catalogue (ASG), User (ASG), Payment (ASG), Dispatch (ASG))
- DB-Subnet (RDS (MySQL), DOCDB (MongoDB), ElastiCache (Redis), RabbitMQ (EC2))

13. ASG (Auto Scaling Group):

An Auto Scaling Group (ASG) in AWS dynamically adjusts the number of EC2 instances based on policies and metrics defined by the user. It ensures applications have enough instances to handle traffic demands efficiently and scales down during low activity to minimize costs. ASGs are integral for maintaining application availability, distributing traffic across multiple instances, and optimizing performance by automatically adjusting capacity in response to changing workloads.

Note: Auto Scaling Groups (ASGs) manage instances based on policies and metrics but do not inherently distribute traffic across them. Load balancers, such as Elastic Load Balancing (ELB), complement ASGs by evenly distributing incoming requests among instances to enhance application availability and scalability.

14. ELB (Elastic Load Balancer):

A load balancer in AWS distributes incoming application or network traffic across multiple targets, such as EC2 instances, containers, or IP addresses, to ensure optimal resource utilization, maximum throughput, and minimal response time. It enhances fault tolerance by rerouting traffic away from unhealthy targets and supports various types of loads balancing strategies, including application, network, and classic load balancing. Load balancers play a critical role in scaling applications horizontally and improving overall application availability and reliability in cloud environments.

Types of Load Balancers in AWS:

1. **Classic Load Balancer (CLB):** Provides basic load balancing across multiple EC2 instances, operating at both the application and network layers. CLBs are suitable for applications that rely on EC2-Classic networks.
2. **Application Load Balancer (ALB):** Best suited for load balancing of HTTP and HTTPS traffic, ALBs operate at the application layer (Layer 7) and route requests based on content across multiple targets, such as EC2 instances or containers.
3. **Network Load Balancer (NLB):** Designed to handle high volumes of TCP and UDP traffic efficiently, NLBs operate at the transport layer (Layer 4). They are capable of handling millions of requests per second while maintaining low latency.
4. **Gateway Load Balancer (GWLB):** A scalable and secure managed service that operates at the edge of the AWS global network. It provides flexible, high-performance load balancing for applications hosted on EC2 instances, containers, and IP addresses.

Note: In our DevOps project we are using ALB (Application Load Balancer) for HTTP/HTTPS traffic.

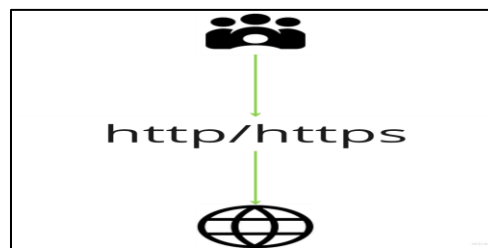


Figure 6

Note: In our DevOps project, we have both Public and Private Load Balancers.

- In the Private Load Balancer, we have a HTTP listener on port 80.
- In the Public Load Balancer, we have HTTP listeners on port 80 and HTTPS listeners on port 443. Additionally, port 80 traffic redirects to port 443 in the Public Load Balancer.

15. TG (Target Group):

A Target Group (TG) in AWS is a logical grouping of targets, such as EC2 instances, containers, or IP addresses, that receive traffic from a load balancer. It enables the load balancer to route requests to registered targets based on configured health checks and routing rules. TGs support dynamic addition and removal of targets, allowing for seamless scaling and high availability of applications. They play a crucial role in distributing incoming traffic efficiently across multiple targets to maintain application performance.

*Note: Auto Scaling Groups (ASGs) in AWS use Target Groups (TGs) for dynamically registering instances. ASGs monitor and adjust the number of instances in the group based on defined policies, while TGs ensure efficient distribution of incoming traffic across the registered instances. This integration allows ASGs to automatically scale capacity in response to demand while ensuring that traffic is evenly distributed to maintain application availability and performance.

16. Route53:

Route 53 is Amazon Web Service's scalable Domain Name System (DNS) service. It translates domain names to IP addresses, directing users to AWS applications. Route 53 supports routing policies like latency-based, weighted, and failover routing. It integrates with AWS services such as CloudFront for content delivery and provides health checks to monitor endpoint health. Route 53 also offers domain registration services, allowing users to search and register domains. It's crucial for reliable, efficient application delivery and DNS management across AWS, ensuring optimal performance and availability for global applications.

*Note: In our DevOps Project we are using Domain Name www.robobal.store

In DNS we are using below record types in our project:

- **NS Records:** Delegate a domain to specific DNS servers responsible for its resolution.
- **SOA Records:** Contain essential domain details, including primary name server and administrator contact.
- **A Records:** Map domain names to specific IPv4 addresses, directing traffic to the correct server.

17. AWS Parameter Store:

AWS Parameter Store, part of AWS Systems Manager, is a secure storage solution for managing configuration data and secrets. It allows you to store data such as passwords, database strings, and license codes as parameter values. Parameters can be plain text or encrypted using AWS Key Management Service (KMS). The service provides a centralized, managed, and versioned store for configuration and secret data, ensuring secure access through fine-grained IAM policies. Integration with other AWS services, like EC2, Lambda, and CloudFormation, enables seamless retrieval of parameters, improving security and simplifying configuration management across your AWS environment.

*Note: In our DevOps project, we store our Dev and Prod environment parameters and secrets in AWS Parameter Store for enhanced authentication and security.

18. AWS KMS (Key Management Service):

AWS Key Management Service (KMS) is a managed service that simplifies the creation and control of cryptographic keys used to encrypt data. It integrates with various AWS services such as S3, EBS, and RDS to provide seamless encryption for data at rest and in transit. KMS offers centralized key management, policy enforcement, and detailed auditing through AWS CloudTrail. It supports both symmetric and asymmetric keys and uses hardware security modules (HSMs) to protect keys. With fine-grained access control, users can define who can use and manage keys, enhancing security and compliance for sensitive data across AWS environments.

In AWS Key Management Service (KMS), there are two types of keys based on management:

1. AWS Managed Keys (AWS KMS-managed keys):

- AWS KMS creates and manages these keys on your behalf.
- These keys are integrated with AWS services for encryption.
- They simplify key management tasks and are ideal for scenarios where you prefer AWS to handle key lifecycle management.

2. Customer Managed Keys (CMKs):

- Customers create, own, and manage these keys themselves within AWS KMS.
- CMKs offer more control over key policies, rotation schedules, and usage.
- They are suitable for compliance requirements or scenarios where stricter control over encryption keys is necessary.

Both types of keys provide strong security through integration with AWS services and can be used to encrypt data stored in AWS or applications running on AWS infrastructure.

In AWS KMS offers two types of cryptographic keys:

1. **Symmetric Keys:** These keys are used for encryption and decryption of data. They are simpler and faster than asymmetric keys, suitable for scenarios where the same key is used for both encryption and decryption operations.
2. **Asymmetric Keys:** Also known as public-private key pairs, asymmetric keys are used for digital signatures and encryption. They consist of a public key, which can be shared openly, and a private key, which is kept secure. Asymmetric keys are typically used in scenarios where secure communication and verification are necessary.

Network and Data Security point of view:

From a network security perspective, we employ private IPs, firewalls, NAT Gateways (NGW), and Internet Gateways (IGW) to connect to our servers securely.

Regarding data security, AWS KMS encrypts sensitive data stored in our AWS Parameter Store, including usernames and passwords. This ensures that access to this information is controlled and managed securely using encryption keys managed by KMS encryption services.

*Note: In our DevOps project we are using Customer Managed Keys (CMKs) with Symmetric Keys.

19. AWS ACM (AWS Certificate Manager):

AWS ACM (AWS Certificate Manager) is a service that simplifies the management of SSL/TLS certificates for websites and applications deployed on AWS. It provides a centralized platform to request, manage, and deploy public and private SSL/TLS certificates. ACM handles certificate provisioning, renewal, and deployment seamlessly, integrating with AWS services like Elastic Load Balancing, CloudFront, and API Gateway. ACM automatically handles certificate renewals, eliminating the operational overhead of manual certificate management and ensuring secure communication between clients and applications.

1.Data at REST we use KMS (Key Management Service)

2.Data at TRANSIT we use SSL (Secure Socket Layer) / TLS (Transport Layer Security)

*Note; Usually for SSL certificate we will approach for third party vendors, but in AWS we can get that as service by using ACM (AWS Certificate Manager).

20. AWS IAM Roles and Policies:

AWS IAM Roles:

AWS Identity and Access Management (IAM) roles are entities with permissions policies that determine what actions can be taken on AWS resources. Roles are used to delegate access to AWS services, applications, or users without the need for long-term credentials. They are commonly used for granting permissions to applications running on AWS EC2 instances, Lambda functions, or to cross-account access.

AWS IAM Policies:

AWS IAM policies are JSON documents that define permissions for IAM identities (users, groups, roles) and AWS resources. Policies specify which actions are allowed or denied, on which resources, and under what conditions. IAM policies can be AWS managed (pre-defined by AWS)

or custom (created by users), and they are attached to IAM identities to control access permissions effectively. Policies enforce security best practices by implementing the principle of least privilege, ensuring that users and services have only the necessary permissions to perform their tasks.

*Note: In our DevOps project, AWS IAM roles and policies are essential for controlling access permissions securely. They ensure that only authorized users and services can interact with AWS resources, maintaining robust security measures throughout our project's operations.

21. AWS AZ (Availability Zones):

An Availability Zone (AZ) is a distinct location within an AWS region, designed to be isolated from failures in other AZs. Each AZ consists of one or more data centers with independent power, cooling, and networking. By distributing applications across multiple AZs, users can achieve high availability and fault tolerance. AZs are connected with low-latency, high-throughput networking, enabling seamless data replication and failover, ensuring resilient and reliable cloud infrastructure for mission-critical applications.

*Note: In our DevOps project, we are using, AZ (Availability Zones) us-east-1a and us-east-1b to ensure high availability and fault tolerance by distributing resources across these zones.

22. AWS RDS (Relational Database Service):

AWS RDS (Relational Database Service) simplifies the setup, operation, and scaling of relational databases in the cloud. It supports various database engines like MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server. RDS handles routine database tasks such as backups, patch management, and hardware provisioning. It offers high availability, scalability, and security features, including automated backups and Multi-AZ deployments. RDS enables developers to focus on application development by managing the complexities of database administration.

*Note: In our DevOps project, we are utilizing an AWS RDS Aurora MySQL 5.7 cluster across multiple Availability Zones (us-east-1a and us-east-1b) on the default port (3306) to ensure enhanced availability and fault tolerance. This service supports the Shipping API by managing schema details for location information efficiently.

23. AWS DOCDB (Document Database):

Amazon Document DB (with MongoDB compatibility) is a fully managed NoSQL database service designed to work with JSON-like document data. It provides seamless scalability, high availability,

and durability with automatic backups and multi-AZ deployments. Document DB offers compatibility with MongoDB workloads, allowing easy migration and integration of existing applications. It simplifies database management by handling tasks such as hardware provisioning, patching, and backups, enabling developers to focus on building applications without worrying about database operations.

*Note: In our DevOps project, we are utilizing an Amazon Document DB (with MongoDB compatibility) cluster across multiple Availability Zones (us-east-1a and us-east-1b) on the default port (27017) to ensure enhanced availability and fault tolerance. This service supports both Catalog API and User API by managing schemas for categories and subcategories efficiently.

24. AWS ElastiCache:

AWS ElastiCache is a fully managed, in-memory caching service compatible with Redis and Memcached. It improves the performance of applications by enabling them to retrieve data from fast, managed, in-memory data stores, instead of relying entirely on slower disk-based databases. ElastiCache supports demanding applications that require low-latency data access and high throughput. It automates routine administrative tasks such as hardware provisioning, setup, patching, and backups, allowing developers to focus on building applications rather than managing infrastructure.

*Note: In our DevOps project, we are leveraging an Amazon ElastiCache cluster (with Redis compatibility) deployed across Availability Zones (us-east-1a and us-east-1b), operating on the default port (6379) to ensure enhanced availability and fault tolerance. This service supports User and Cart APIs by enabling fast data retrieval, caching user profiles, managing sessions, facilitating real-time updates, and scaling efficiently. Redis's in-memory storage and pub/sub messaging optimize performance, ensuring rapid access to essential data and timely updates for shopping carts, thereby enhancing user experience.

25. RabbitMQ (Deployed on AWS EC2 Instances):

RabbitMQ is a robust, open-source message broker that facilitates communication between distributed systems by implementing the Advanced Message Queuing Protocol (AMQP). It ensures reliable message delivery, supports multiple messaging protocols, and offers features like message queuing, routing, and clustering. RabbitMQ is widely used in microservices architectures and other applications where asynchronous communication is critical for decoupling components and scaling systems efficiently. It enables seamless integration and communication between various components of modern distributed applications.

*Note: In our DevOps project, we utilize RabbitMQ deployed on AWS EC2 instances across Availability Zones (us-east-1a and us-east-1b), operating on port 5672. RabbitMQ supports Payment and Dispatch services by enabling reliable message queuing, routing, and asynchronous communication, ensuring efficient processing and delivery of payment and dispatch-related messages.

Project VPC structure:

The below diagram is our DevOps project VPC structure with all the AWS services with the workflows and connectivity. Refer these URLs in MIRO for better understanding of VPC structure:

https://miro.com/app/board/uXjVK5A1-I4=/?share_link_id=483473893870

<https://miro.com/app/board/uXjVK5A1-I4=/?fromEmbed=1>

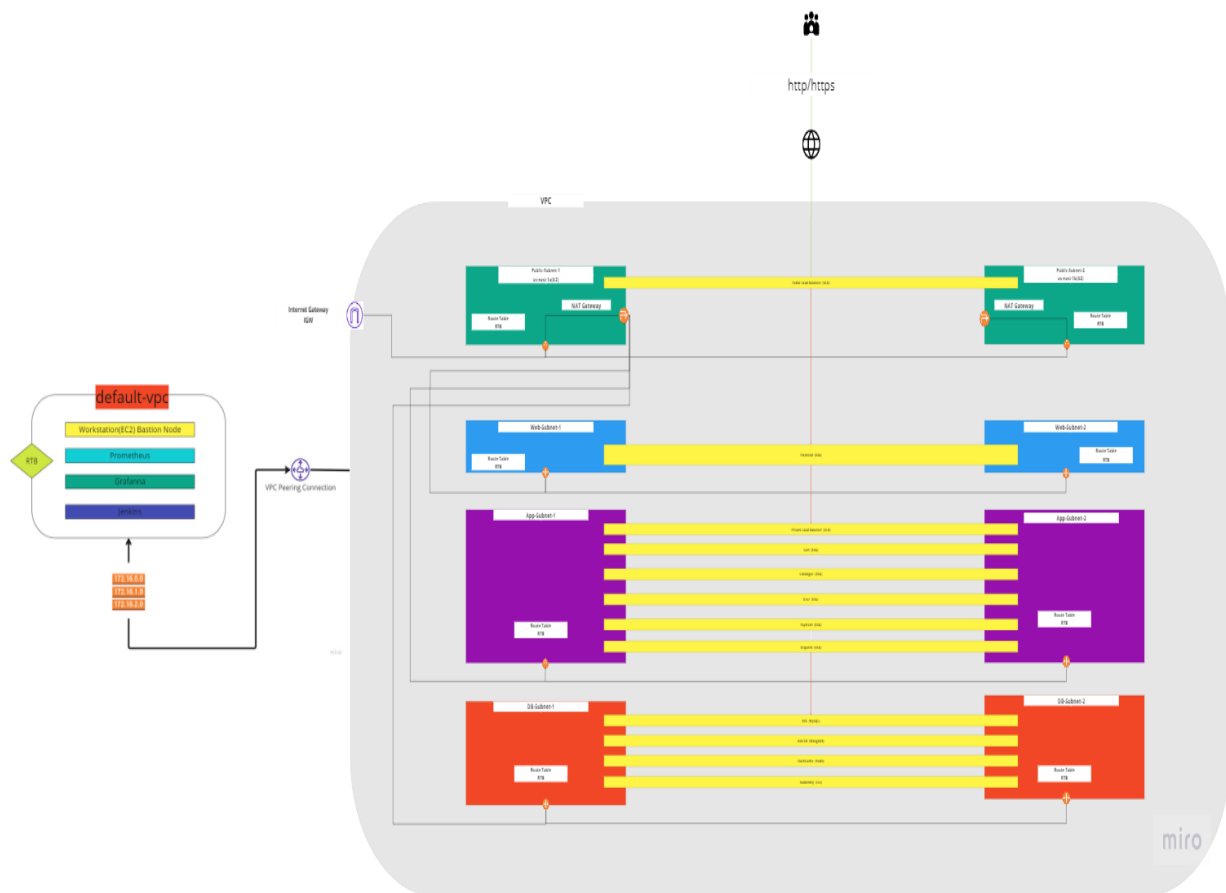


Figure 7

Jenkins:

Jenkins is an open-source automation server extensively used for continuous integration and continuous delivery (CI/CD) in software development. It automates build, test, and deployment processes, allowing developers to detect and resolve issues early. Jenkins supports a wide range of plugins, making it highly customizable and adaptable to various development environments. It integrates with numerous tools such as Git, Maven, AWS, Ansible, SonarQube, Nexus, Terraform, Docker, and Kubernetes, facilitating seamless workflow automation. By automating repetitive tasks and providing real-time feedback, Jenkins enhances productivity, accelerates development cycles, and improves software quality. Its robust community support and extensive documentation make it a popular choice for DevOps teams.

2.1 CI/CD (Continuous Integration & Continuous Deployment):

CI/CD (Continuous Integration and Continuous Deployment) is a crucial practice in modern software development.

CI (Continuous Integration): Plan, Code, Build and Test stages/scenarios:

Continuous Integration involves automatically merging and testing code changes from multiple contributors, ensuring early detection of integration issues. Developers frequently commit code to a shared repository, triggering automated builds and tests.

CD (Continuous Deployment): Release, Deploy and Operate:

Continuous Delivery extends this by automatically deploying tested code to production environments, ensuring that software can be reliably released at any time. This automation reduces manual errors, accelerates release cycles, and enhances software quality.

CI/CD fosters a collaborative development culture, enabling teams to deliver features and updates faster, with greater confidence and efficiency, ultimately improving the end-user experience.

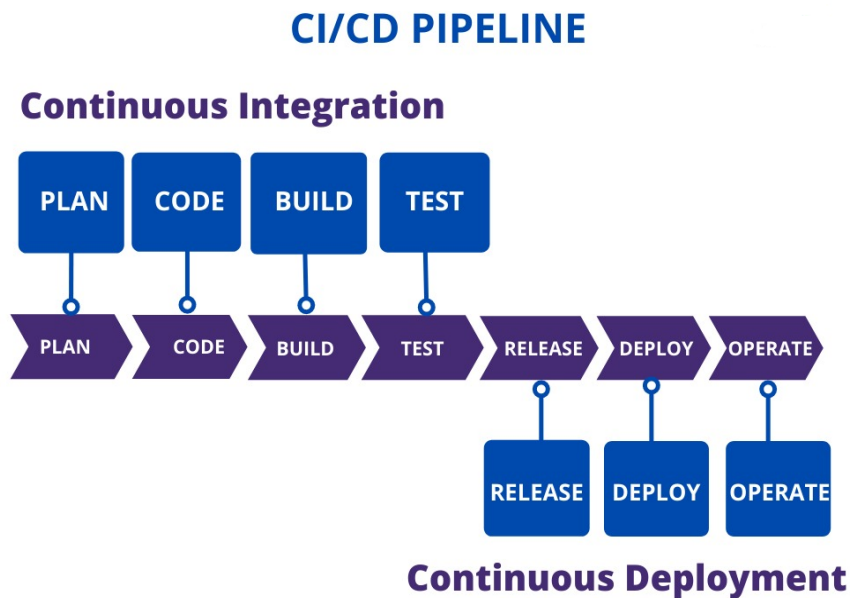


Figure 8

2.2. Jenkins Architecture:

Jenkins typically supports the following types of architectures:

1. Single-Node Architecture

- **Components:**
 - **Master Node:** Handles all Jenkins functionalities including job scheduling, build execution, and plugin management.
 - **Build Executors:** Execute jobs directly on the master node.
- **Use Case:** Suitable for small projects or environments with minimal build and deployment requirements.

2. Multi-Node (Master-Agent) Architecture

- **Components:**
 - **Master Node:** Manages the overall Jenkins operations, job scheduling, and assignment of build tasks to agents.
 - **Agent Nodes:** Execute the build jobs assigned by the master node.
- **Use Case:** Ideal for larger projects needing scalability, parallel job execution, and load distribution.

3. Distributed Architecture

- **Components:**
 - **Master Node:** Orchestrates build tasks and manages the configuration.
 - **Multiple Agent Nodes:** Distributed across various machines or data centers, executing build tasks in parallel.
- **Use Case:** Used for very large, complex projects requiring extensive resources, fault tolerance, and geographic distribution.

4. Cloud-Based Architecture

- **Components:**
 - **Master Node:** Runs on a cloud instance, managing job scheduling and build orchestration.
 - **Cloud-Based Agent Nodes:** Dynamically provisioned and de-provisioned based on the workload, executing build tasks in the cloud.
- **Use Case:** Suitable for environments requiring dynamic scaling and resource optimization based on demand.

5. Hybrid Architecture

- **Components:**
 - **Master Node:** Manages builds, typically running on-premises.
 - **Hybrid Agent Nodes:** Combination of on-premises and cloud-based agents executing build tasks.
- **Use Case:** Used when there is a need to leverage both on-premises resources and cloud capabilities for flexibility and scalability.

Each architecture type can be chosen based on the specific needs, scalability requirements, and infrastructure constraints of the project.

Limitations of Single-Node Architecture:

1. Jenkins is typically deployed as a standalone software, meaning it runs on a single machine.
2. Jobs also execute on this instance, requiring all necessary software (e.g., Terraform, Ansible) to be installed on the same machine.
3. This setup poses a risk: if the Jenkins machine fails, all running jobs fail, leading to a single point of failure.

Solution: To mitigate risks, Jenkins can be configured to operate in a multi-node architecture. In this setup, job execution is distributed across multiple machines, reducing the impact of a single node failure.

Additional Needs: A single Jenkins server may not suffice for environments requiring multiple distinct testing environments or handling large and resource-intensive projects efficiently. Scaling to multiple nodes addresses these challenges by distributing workload and enhancing overall system resilience and performance.

*Note: In our DevOps Project we are using Distributed Master-Slave Architecture.

Jenkins Distributed Master-Slave Architecture:

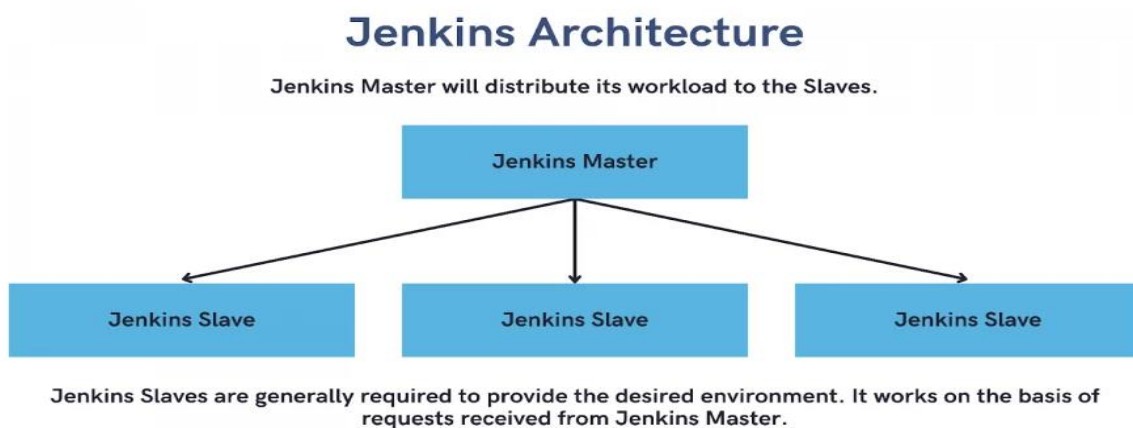


Figure 9

Jenkins Master: The primary Jenkins server responsible for:

- Scheduling build jobs.
- Recording and presenting build results.
- Distributing builds to slaves for execution.
- Monitoring slave status (online/offline).

Jenkins Slave: Remote servers that:

- Execute build jobs assigned by the master.

- Support multiple operating systems.
- Can be configured to execute specific project tasks.

This architecture enhances scalability, flexibility, and resilience in Jenkins environments. Jenkins manages distributed builds using the Master-Slave architecture. The TCP/IP protocol is used to communicate between the Master and Slave architecture.

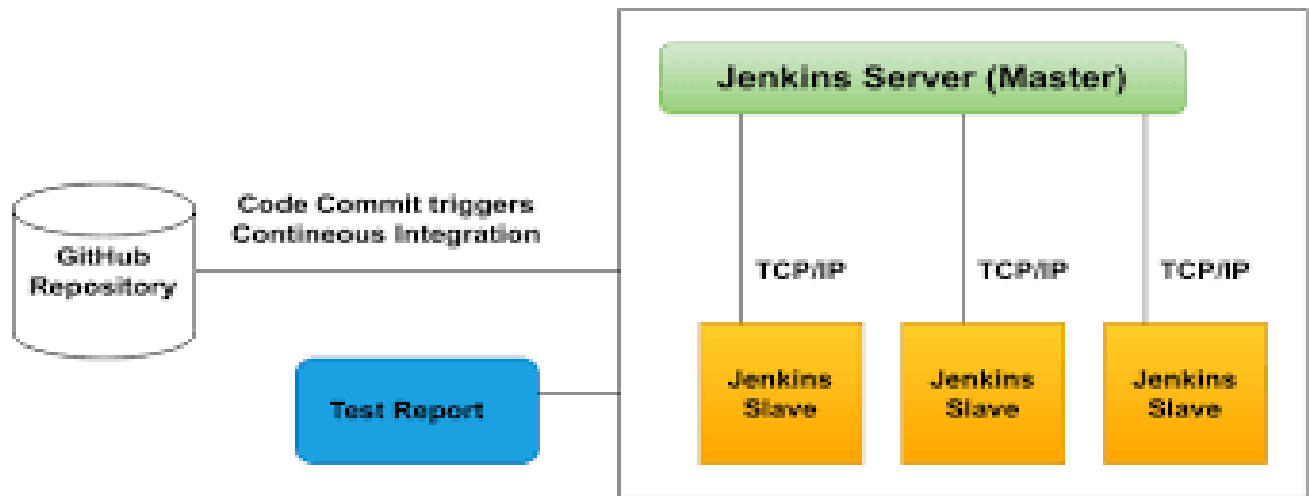


Figure 10

- The workload of the Jenkins Master is distributed to the slaves.
- Jenkins Slaves provide a working environment and operate based on requests from the Jenkins Master.
- Jenkins checks the GIT repository for changes in the source code at regular intervals.
- Each Jenkins build requires its own testing environment, which cannot be created on a single server. Jenkins accomplishes this by employing various slaves as needed.
- The Jenkins Master communicates testing requests and receives test reports from the slaves.

Jenkins Built-in-Node Automation:

OS-Level Automation with Ansible

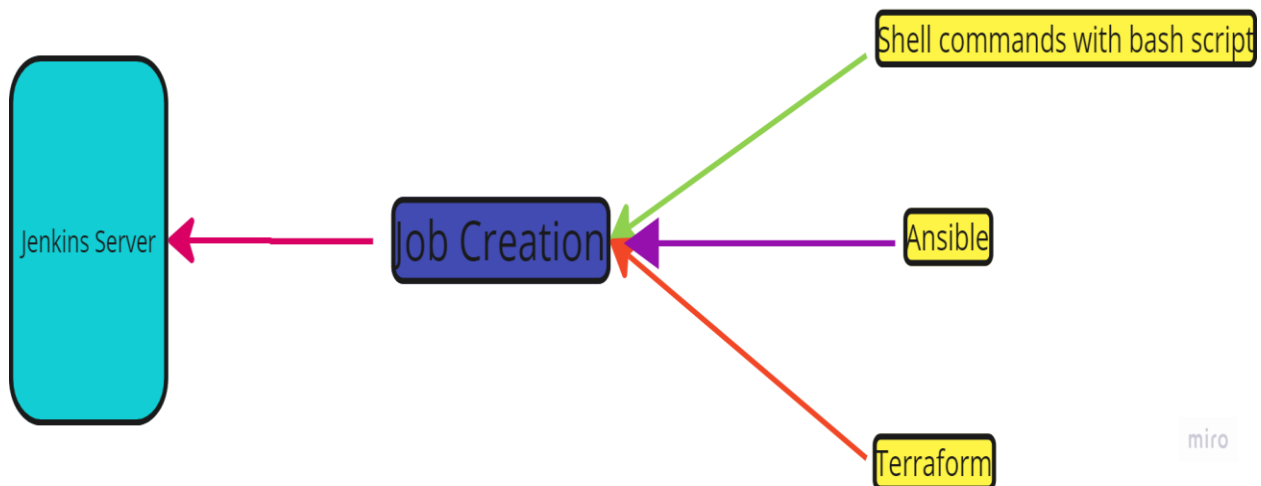
For **OS-level automation**, Jenkins leverages **Ansible**, allowing the automation of software provisioning, configuration management, and application deployment across different environments.

Cloud-Level Automation with Terraform

At the **cloud infrastructure level**, **Terraform** is utilized for **Infrastructure as Code (IaC)**, enabling the automated setup, modification, and versioning of cloud resources.

Comprehensive Automation Approach

This comprehensive automation approach ensures that every aspect, from code deployment to infrastructure management, is efficiently handled, reducing manual intervention, minimizing errors, and enhancing overall productivity and scalability in complex DevOps environments.



*Note: In our DevOps project:

- All jobs will be automated using shell scripts (bash).
- All OS (Operating System) level automations will be done using Ansible.
- All cloud (infrastructure) level automations will be done using Terraform.

Our requirement is to avoid running jobs on our Jenkins Master Server (Master). Instead, we want to use the Jenkins Agent Server (Slave), which in our case is the "Workstation" (bastion server).

- **Master:** Jenkins Main Server – Controls management tasks.
- **Slave:** Jenkins Agent Server – Handles compute tasks.

Jenkin Jobs:

In our DevOps project we are using three (3) types of jobs they are:

1. Seed Jobs (Free Style Jobs):

A seed job in Jenkins is a type of job that dynamically generates other jobs based on predefined templates or configurations. It automates the creation of Jenkins jobs by defining them in code or scripts, ensuring consistency and efficiency in managing and deploying pipelines and jobs across various projects or environments.

Using Seed Job in Ansible way for Jenkins Job Automation:

A seed job in Jenkins serves as a template or generator for creating multiple jobs automatically based on predefined configurations. Using Ansible to automate seed jobs enhances efficiency by ensuring consistent job creation across environments. It enables version-controlled management of job configurations, reducing manual errors and enhancing reproducibility in CI/CD pipelines. Ansible's automation capabilities allow for the seamless integration of Jenkins job creation into broader infrastructure management workflows. This approach not only simplifies the initial setup of jobs but also facilitates ongoing maintenance and updates, ensuring that changes to job configurations are applied uniformly and reliably across the Jenkins environment.

2. Single Pipeline Jobs:

Single pipeline jobs in Jenkins represent a straightforward CI/CD approach where each job corresponds to a single pipeline definition. They are ideal for projects with linear workflows or those requiring a sequential execution of stages such as build, test, and deploy. Single pipeline jobs offer simplicity in configuration and execution, making them easier to manage and debug. However, they may not scale well for complex projects that require parallel execution of tasks or have diverse dependencies across different stages of development.

Our Single Pipeline Jobs (sp_jobs) are:

1. roboshop-ansible
2. roboshop-terraform
3. roboshop-jenkins.

3. Multi Pipeline Jobs:

Multipipeline jobs in Jenkins allow for the management of multiple related pipelines within a single job configuration. This approach is beneficial for complex projects with interconnected workflows across different components or modules. Each pipeline within the multipipeline job can have its own stages, dependencies, and execution logic, enabling parallel execution of tasks and efficient resource utilization. Multipipeline jobs enhance flexibility, scalability, and maintainability by organizing related pipelines under a unified configuration, facilitating coordinated development and deployment processes across the software lifecycle.

Our Multi Pipeline Jobs (mp_jobs) are:

cart

catalogue

user

shipping

payment

frontend

aws-parameter-init-container

schema-load-init-container

Project DevOps CI/CD Pipeline:

1. Code Checkout from SCM (Git/GitHub)

The pipeline starts by checking out the latest code from the source control management system, which in this case is Git or GitHub. This ensures that the pipeline operates on the most recent version of the codebase.

2. Build with Maven

Once the code is checked out, Maven is used to build the project. This involves compiling the source code, resolving dependencies, and packaging the application into a deployable format, such as a JAR or WAR file.

3. Unit Tests

After building the project, unit tests are executed to verify that individual components of the application are functioning as expected. These tests are automated and provide immediate feedback on the quality of the code.

4. Code Analysis with SonarQube

SonarQube is then used to perform static code analysis. This stage checks for code quality issues, such as code smells, bugs, and duplications. It helps in maintaining high standards of code quality throughout the development process.

5. Security Check with SonarQube

In addition to code quality analysis, SonarQube is also used to perform security checks. This stage identifies vulnerabilities and security hotspots in the code, helping to ensure that the application is secure and robust against potential threats.

6. Publish an Artifact with Version (Nexus/AWS ECR)

Once the code passes the unit tests and analysis stages, the build artifact is published to a repository like Nexus or AWS Elastic Container Registry (ECR). The artifact is versioned to keep track of different releases and facilitate deployment.

7. Release with a Tag

The next stage involves tagging the release in the source control system. This tag serves as a snapshot of the codebase at the point of release, making it easier to reference and rollback if necessary.

8. Deploy to AWS (ELK)

The final stage is deploying the application to AWS environments using ELK (Elastic Load Balancing, EC2, and IAM). This deployment is managed across three environments: development (dev), quality assurance (QA), and production (prod).

Deployment Environments

Dev-Env

The application is first deployed to the development environment, where developers can perform initial testing and integration.

QA-Env

After successful deployment and testing in the development environment, the application is deployed to the QA environment for further testing. This includes functional, performance, and regression testing by the QA team.

Prod-Env

Finally, the application is deployed to the production environment, where it becomes accessible to end-users. This deployment is usually the last step after all tests have been passed successfully in the previous environments.

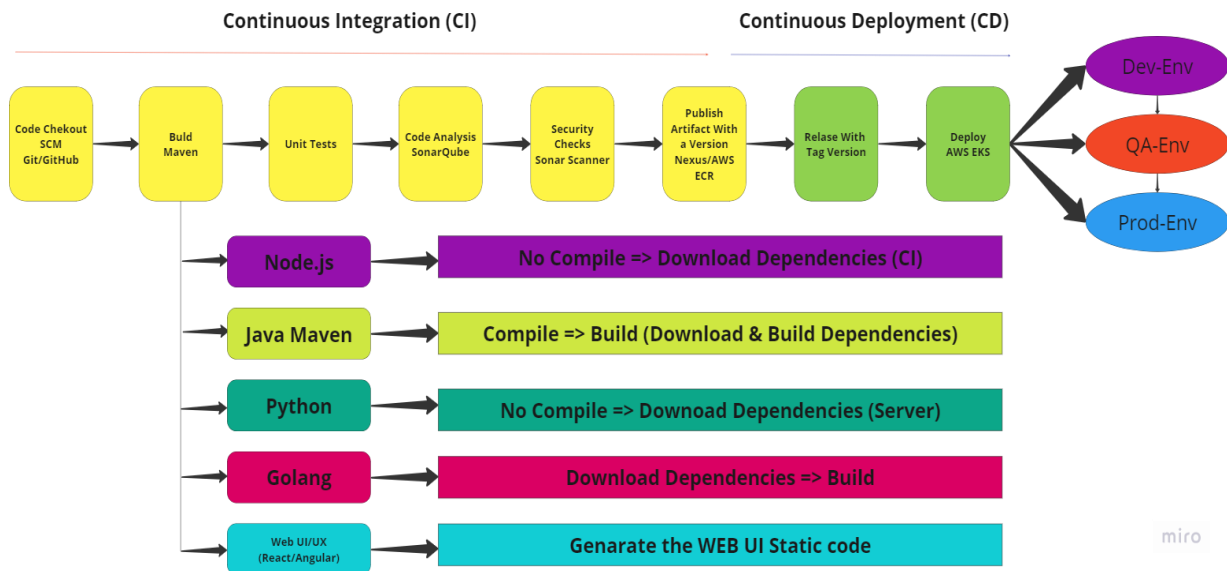


Figure 11

*Note: In our DevOps project we are using the CI/CD for Dev-Env and Prod-Env.

Seed Job Dynamically building Single Pipeline & Multi Pipeline jobs using Jenkins and Ansible.

S	W	Name ↓	Last Success	Last Failure	Last Duration	Fav
🟢	☀️	aws-parameter-init-container	2 days 23 hr log	N/A	0.58 sec	▶️ ⭐
🟢	☀️	cart	1 day 23 hr log	N/A	0.87 sec	▶️ ⭐
🟢	☀️	catalogue	1 day 23 hr log	N/A	0.73 sec	▶️ ⭐
🟢	☀️	frontend	18 min log	N/A	0.59 sec	▶️ ⭐
🟢	☀️	payment	1 day 23 hr log	N/A	0.61 sec	▶️ ⭐
🟢	☀️	roboshop-ansible	16 days #22	18 days #17	1 min 32 sec	▶️ ⭐
🟢	☀️	roboshop-kubernetes	17 min #86	2 days 0 hr #61	5.1 sec	▶️ ⭐
🔴	☁️	roboshop-terraform	N/A	2 days 0 hr #2	0.29 sec	▶️ ⭐
🟢	☀️	sample	26 days log	N/A	0.85 sec	▶️ ⭐
🟢	☀️	seed	2 days 23 hr #11	26 days #6	22 sec	▶️ ⭐

Figure 12

Jenkins CI (Continuous Integration) Pipeline

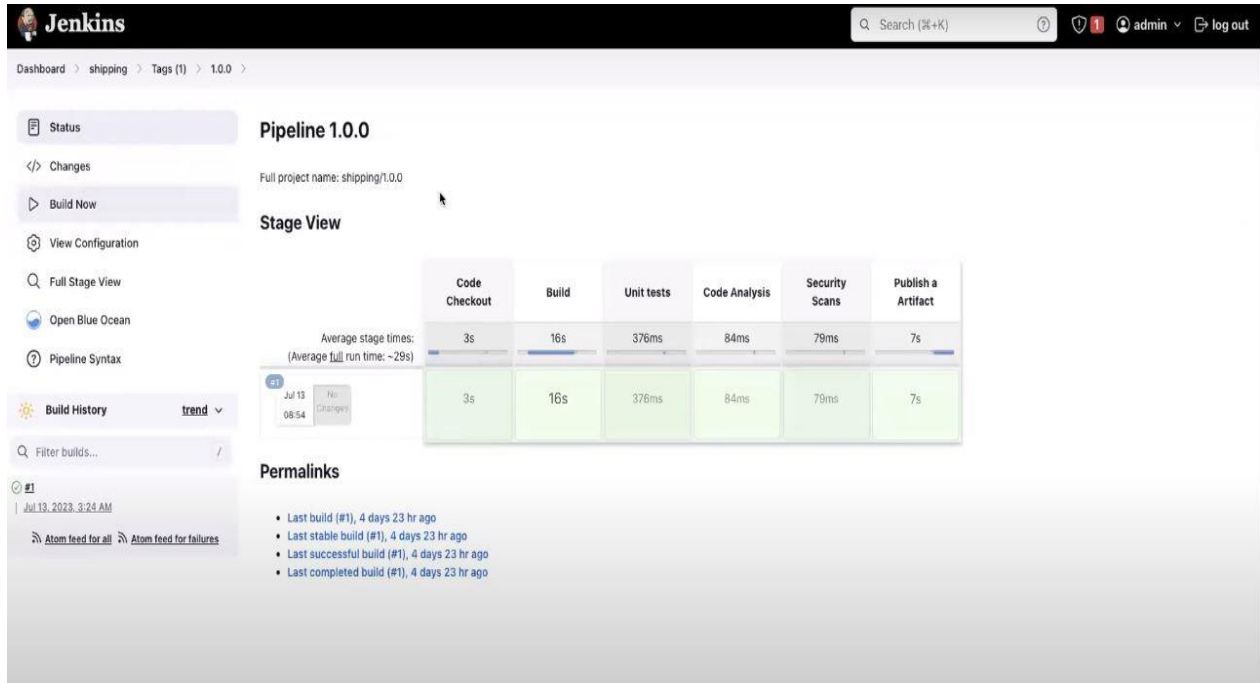


Figure 13

Deploying components/microservices in Dev & Prod Environments using App_Versions (Tags).

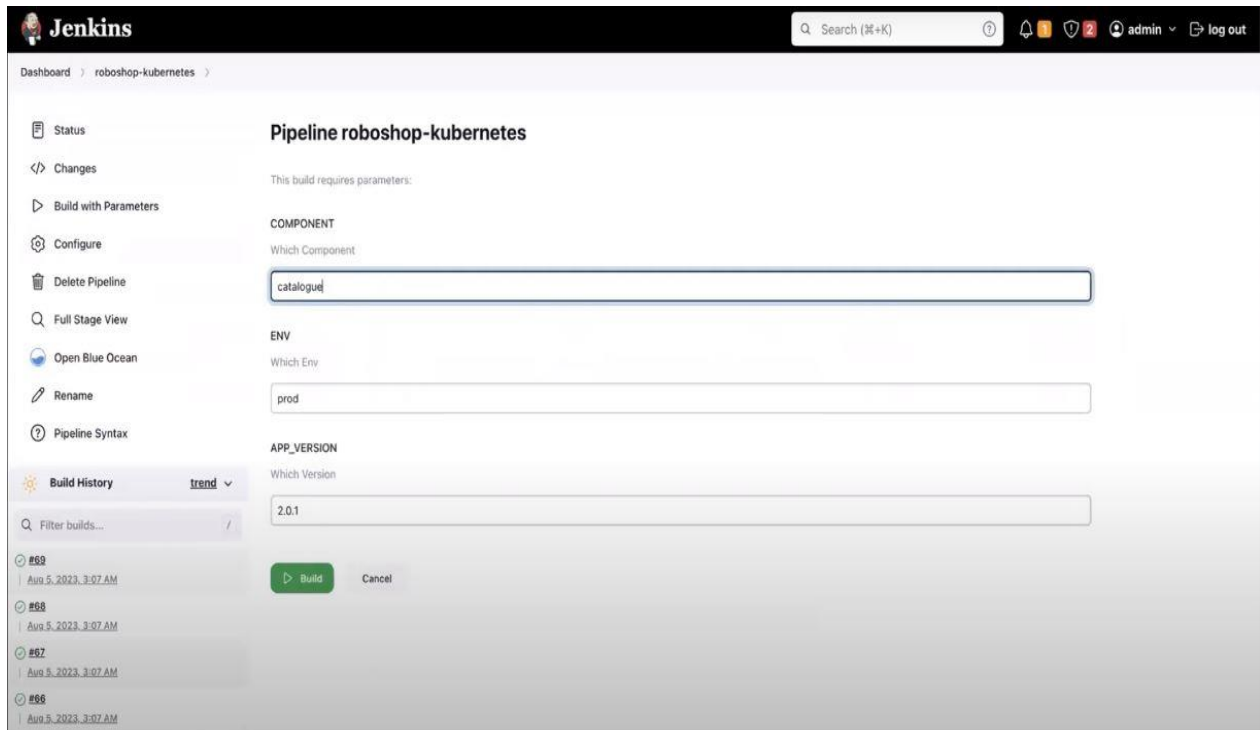


Figure 14

Jenkins CD (Continuous Deployment) Pipeline

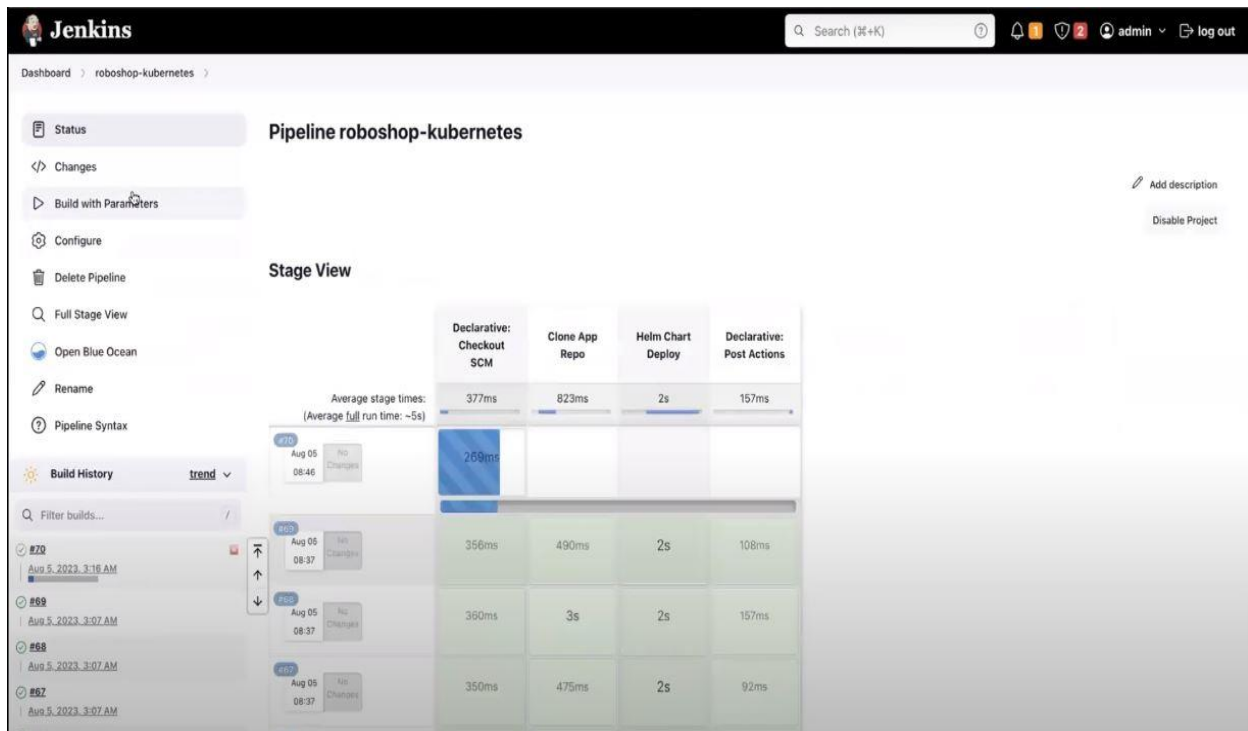


Figure 15

3. SonarQube:

SonarQube is an open-source platform designed for continuous inspection of code quality and security. It performs comprehensive static code analysis to detect code smells, bugs, and duplications, which helps maintain and improve the overall quality of the codebase. By identifying these issues early, developers can prevent potential problems and ensure the code is clean, efficient, and maintainable.

In addition to code quality, SonarQube is equipped to perform security analysis. It identifies vulnerabilities and security hotspots, allowing developers to address potential threats and fortify the application against attacks. This dual focus on code quality and security makes SonarQube an invaluable tool for development teams aiming for high standards.

SonarQube supports a wide range of programming languages, making it versatile for diverse development environments. It integrates seamlessly with CI/CD pipelines, such as Jenkins, GitLab CI/CD, and others, providing automated and continuous feedback on code quality and security with every code change.

Developers can leverage SonarQube's detailed dashboards and reports to track progress over time, prioritize issues, and ensure that the code meets predefined quality gates before deployment. By incorporating SonarQube into the development process, teams can enhance their productivity, reduce technical debt, and deliver more secure and reliable software.

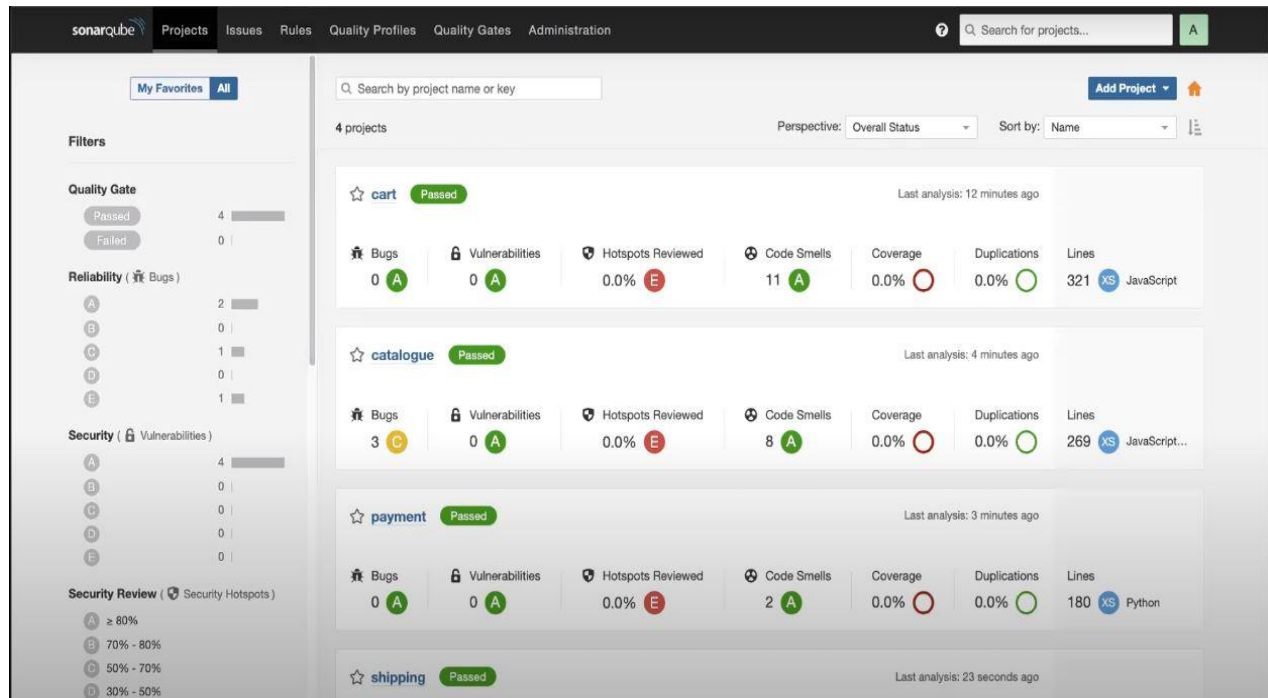


Figure 16

*Note: In our DevOps project we are using separate AWS EC-2 Instance and installing this SonarQube on this instance/server for code analysis.

Sonar Scanner:

Sonar Scanner is a key component of the SonarQube ecosystem, designed to perform code analysis and send the results to the SonarQube server. It is a versatile and flexible tool that supports multiple programming languages, enabling thorough inspection of codebases. Sonar Scanner operates by parsing the source code, identifying code smells, bugs, duplications, and security vulnerabilities.

The tool integrates seamlessly with various build systems, including Maven, Gradle, Ant, and others, making it easy to incorporate into existing CI/CD pipelines. Developers can run Sonar Scanner manually or configure it to run automatically during the build process. The results are then displayed on the SonarQube dashboard, where detailed reports provide insights into code quality and security issues.

Sonar Scanner's ability to work with different development environments and its ease of integration make it an essential tool for maintaining high standards of code quality and security, ultimately helping teams to deliver more reliable and maintainable software.

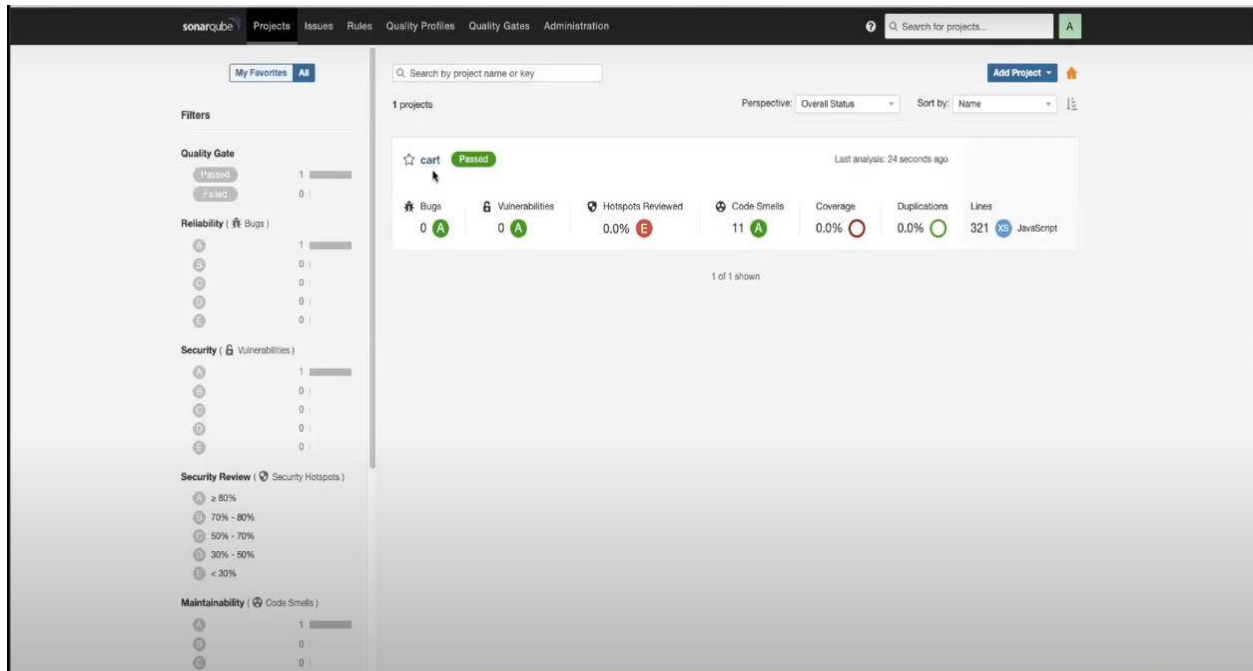


Figure 17

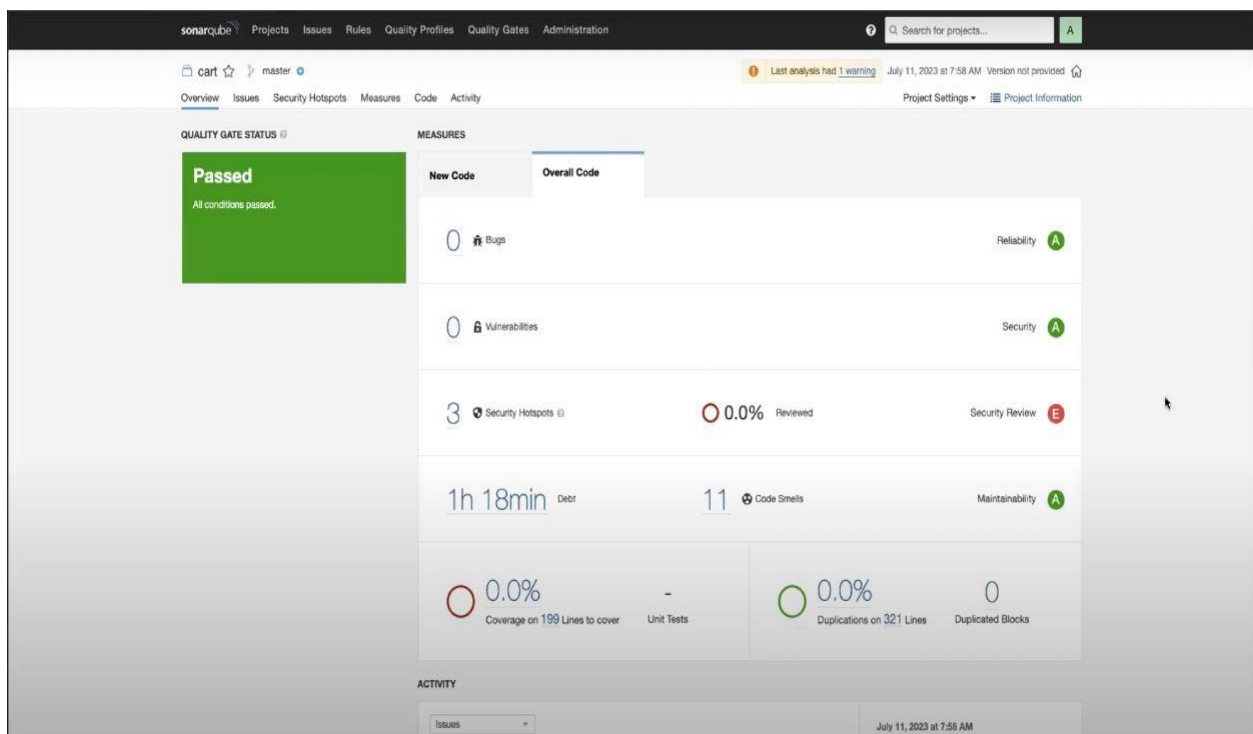


Figure 18

*Note: We are installing Sonar Scanner on our "Workstation," which serves as our bastion server since all our jobs run there. Sonar Scanner primarily analyzes the code and sends the results to the specified SonarQube server. The SonarQube quality gate ensures code quality by defining criteria; if these criteria are met, the code passes. If not, developers must address these issues before approval, thereby maintaining high software standards and reliability.

4. Nexus:

Nexus simplifies the management of binaries and dependencies, enabling development teams to efficiently store, version, and retrieve artifacts during the build process.

Nexus can be integrated into CI/CD pipelines, allowing automated storage and retrieval of build artifacts, which enhances the efficiency and reliability of the software development lifecycle. It also supports proxying of remote repositories, caching artifacts locally to save bandwidth and improve build performance.

Security features in Nexus include fine-grained access control, ensuring that only authorized users can access and deploy artifacts. Additionally, it offers auditing and logging capabilities for tracking artifact usage and modifications.

By centralizing artifact management, Nexus helps teams maintain consistent and reproducible builds, reduce build times, and improve collaboration, making it a vital component in modern DevOps practices.

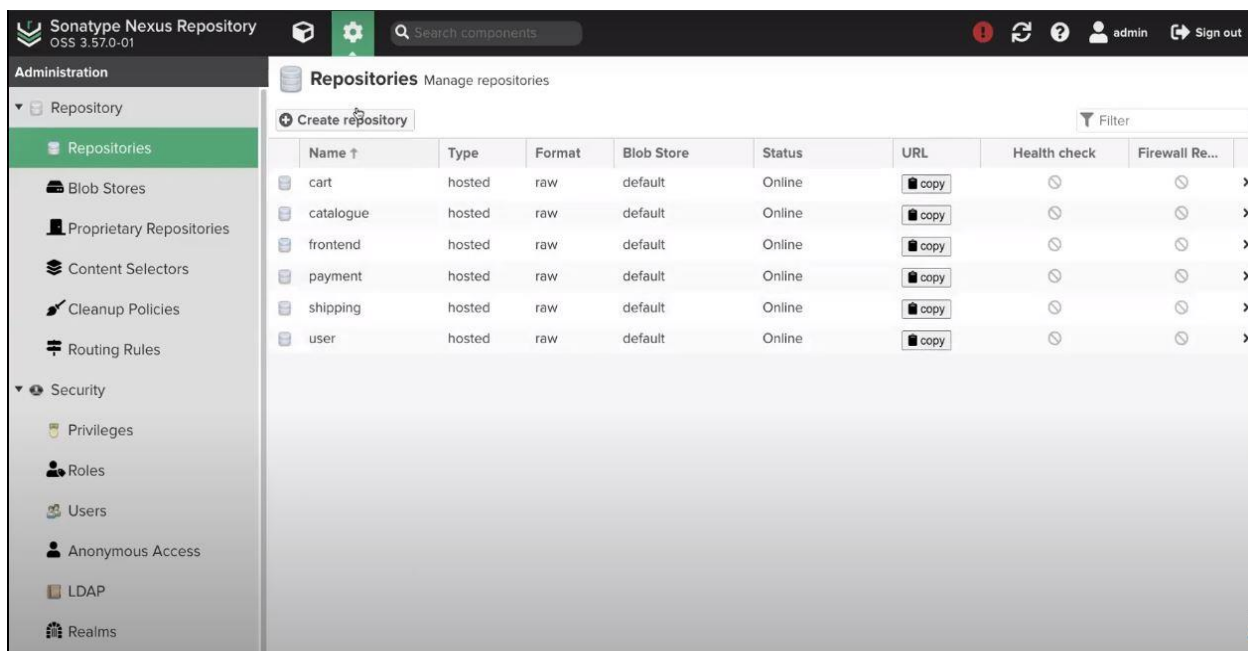


Figure 19

*Note: In our DevOps project we are using separate AWS EC-2 Instance and installing this Sonatype Nexus Artifacts Repository on this instance/server for storing the artifacts with different versions.

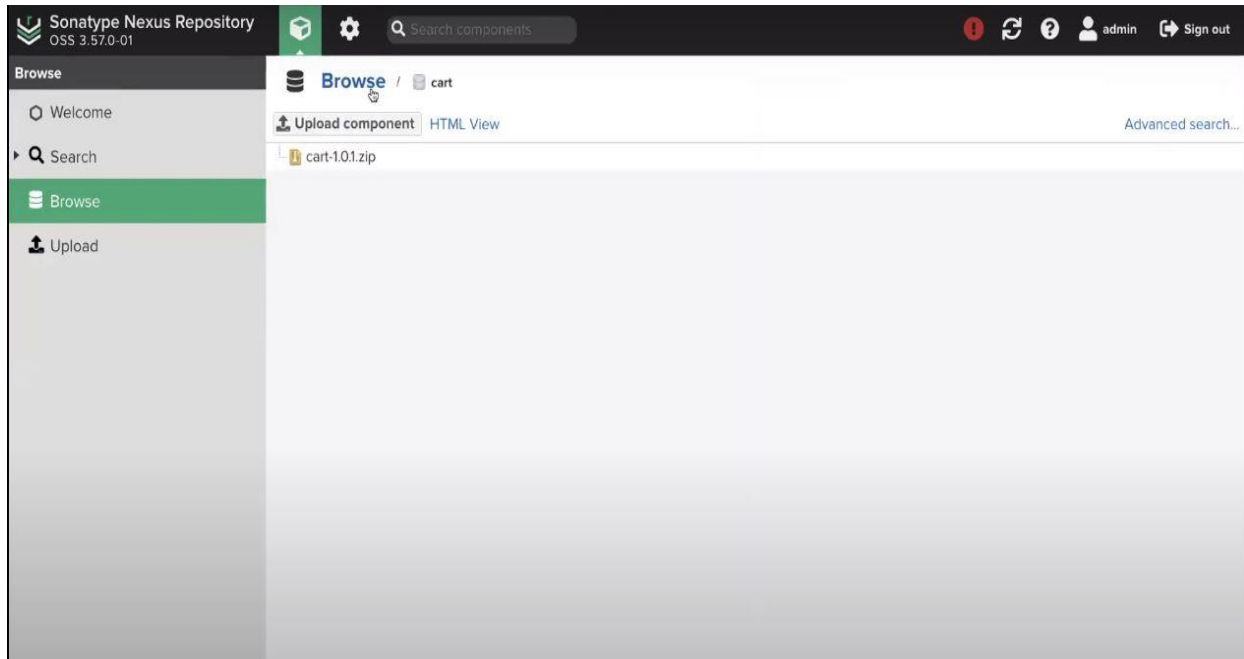


Figure 20

*Note: Previously, we downloaded the application content from an AWS S3 bucket. However, we are now transitioning to fetch artifacts from the Nexus Repository. This change will streamline our artifact management process, improve version control, and enhance the efficiency of our CI/CD pipeline by utilizing Nexus's robust features.

5.Git/ GitHub (SCM):

Git is a powerful distributed version control system that tracks changes in source code, enabling multiple developers to collaborate without conflicts. Key features include branching, merging, and history tracking, which streamline project management and development workflows for both small and large-scale projects.

GitHub, built on top of Git, is a cloud-based platform that extends Git's functionalities by providing tools for collaboration, code review, and project management. It supports pull requests, allowing developers to propose, discuss, and review code changes before integration. GitHub issues and project boards help teams manage tasks and track progress efficiently.

Integration with various development and CI/CD tools makes GitHub a central hub for automation, enhancing the software development lifecycle. Features like GitHub Actions facilitate automated testing and deployment, improving code quality and accelerating delivery.

Together, Git and GitHub offer a comprehensive source code management (SCM) solution, essential for modern DevOps practices. They enable seamless development, continuous integration, and collaborative efforts, making them indispensable tools in the software development industry.

Git/GitHub Workflow in Local and Remote Servers.

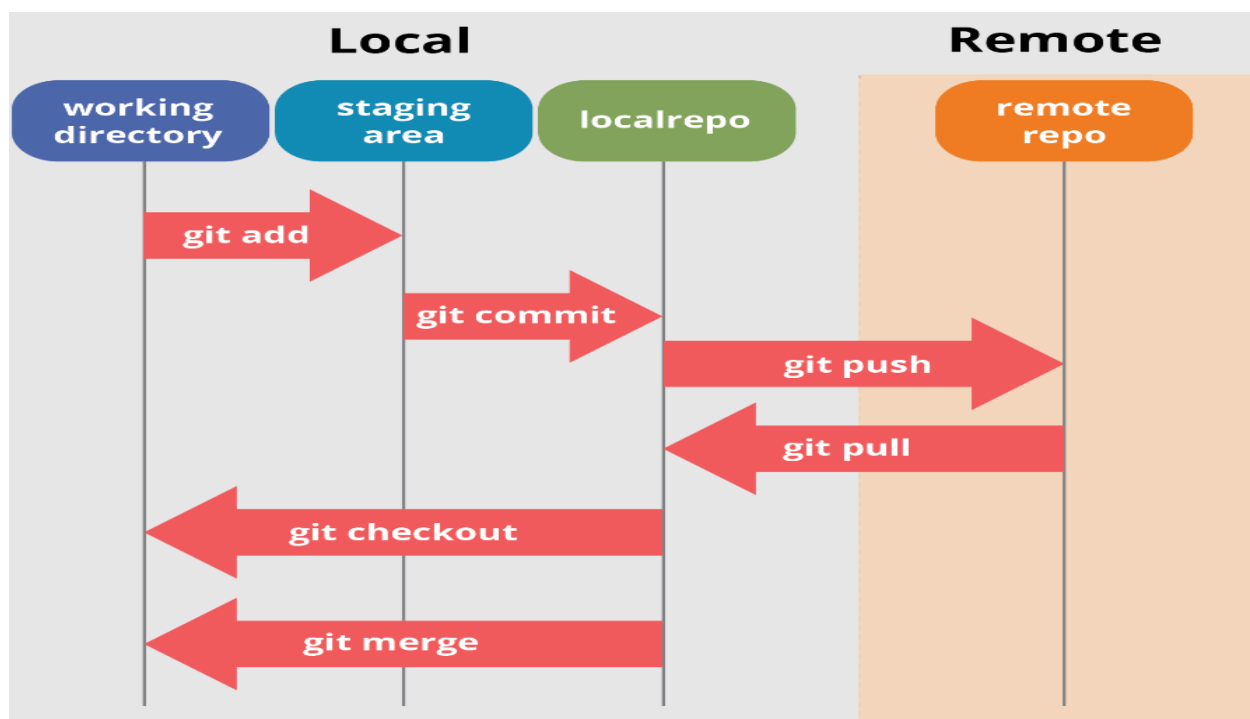


Figure 21

GitHub Webhooks:

GitHub webhooks are a powerful feature that allows you to automate workflows and integrate with external services by sending HTTP POST payloads to specified URLs when certain events occur in a repository. This can be used for a variety of purposes, including triggering CI/CD pipelines, updating external tools, or automating tasks based on repository activity.

Here is an overview of how GitHub webhooks work:

1. **Setup:** You configure a webhook in your GitHub repository settings by specifying a payload URL (where the POST requests will be sent) and selecting which events should trigger the webhook (e.g., push, pull request, issues).
2. **Payload:** When an event you have subscribed to occurs, GitHub sends a payload to the specified URL. This payload contains detailed information about the event, such as the type of event, the repository it occurred in, and data relevant to the event (e.g., commit details for a push event).
3. **Processing:** Your server or application listens for these POST requests at the payload URL, processes the received data, and performs actions accordingly. For example, it might trigger a Jenkins build, update a Slack channel, or log the event in a monitoring system.
4. **Security:** To secure webhooks, you can configure a secret token that GitHub includes in the headers of each payload. Your server can then verify this token to ensure that the requests are indeed from GitHub.

GitHub webhooks facilitate seamless integration and automation, making them an essential tool for modern DevOps and continuous integration/continuous deployment (CI/CD) workflows.

*Note: In our DevOps project:

- We are using Git/GitHub as the single source of truth for our entire application codebase.
- Once the project passes all phases in CI (Continuous Integration), we release the project software using tagging and deploy into desired environments (Dev, QA, Prod) CD (Continuous Deployment). In Jenkins pipelines, tagging is vital for associating version identifiers (tags) with specific commits, marking stable points in the codebase. This enhances version control and aids organized release management. Tagging during artifact publication ensures traceability and efficiency, contributing to a well-managed CI/CD software release process.
- With the help of GitHub webhooks, we automate our Jenkins CI/CD pipelines. When a new commit with a new tag is pushed, the webhook triggers the build and release of the software automatically into desired environments (Dev, QA, Prod).

6. Ansible:

Ansible is an open-source automation tool used for configuration management, application deployment, and task automation. It simplifies complex tasks by using a declarative language (YAML) to define system configurations, software installations, and other IT tasks. Ansible operates on an agentless architecture, meaning it does not require any software to be installed on the target machines, instead relying on SSH for Linux or WinRE for Windows to execute commands.

Key features of Ansible include:

1. **Simplicity and Ease of Use:** Ansible uses simple, human-readable YAML files (known as playbooks) to define automation tasks. This makes it accessible to both developers and operations teams, promoting collaboration and efficiency.
2. **Agentless Architecture:** Ansible's agentless nature eliminates the need for managing agents on target machines, reducing overhead and simplifying maintenance.
3. **Idempotency:** Ansible ensures that tasks are repeatable and can be executed multiple times without changing the system state if it's already in the desired state.
4. **Extensibility:** Ansible provides a modular framework where you can write custom modules in Python to extend its capabilities.
5. **Scalability:** Ansible can manage large-scale environments, supporting complex deployments and configurations across thousands of machines.
6. **Integration:** Ansible integrates seamlessly with other tools and platforms, including cloud providers like AWS, Azure, and Google Cloud, making it suitable for managing hybrid and multi-cloud environments.

Ansible is a versatile and powerful tool that automates IT tasks, ensuring consistency, efficiency, and scalability across diverse environments, and is a cornerstone in modern DevOps practices.

Ansible Architecture:

Ansible operates on an agentless architecture, relying on a central control node to manage target machines (hosts) via SSH for Linux/Unix systems and WinRM for Windows systems. The core components include:

1. **Control Node:** Executes Ansible playbooks, which are YAML files defining automation tasks.
2. **Managed Nodes:** The target machines being configured and managed.
3. **Inventory:** Lists of managed nodes and groups, specifying their connection details.
4. **Modules:** Executable code units on managed nodes performing specific tasks.
5. **Playbooks:** YAML files containing organized sets of tasks.

This architecture simplifies deployment, reduces overhead, and ensures consistency across environments.

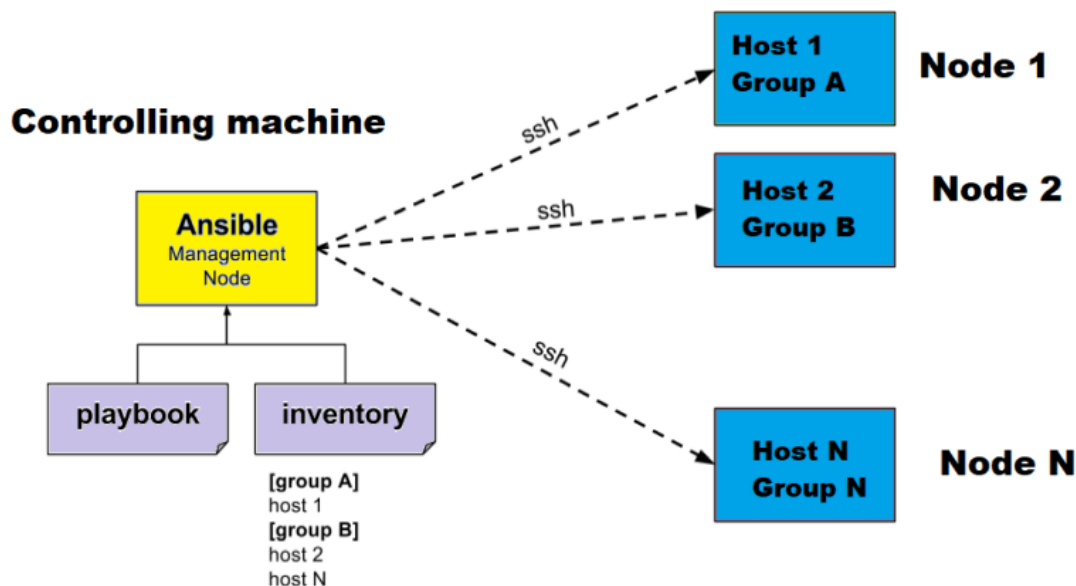


Figure 22

Operational Models in Ansible:

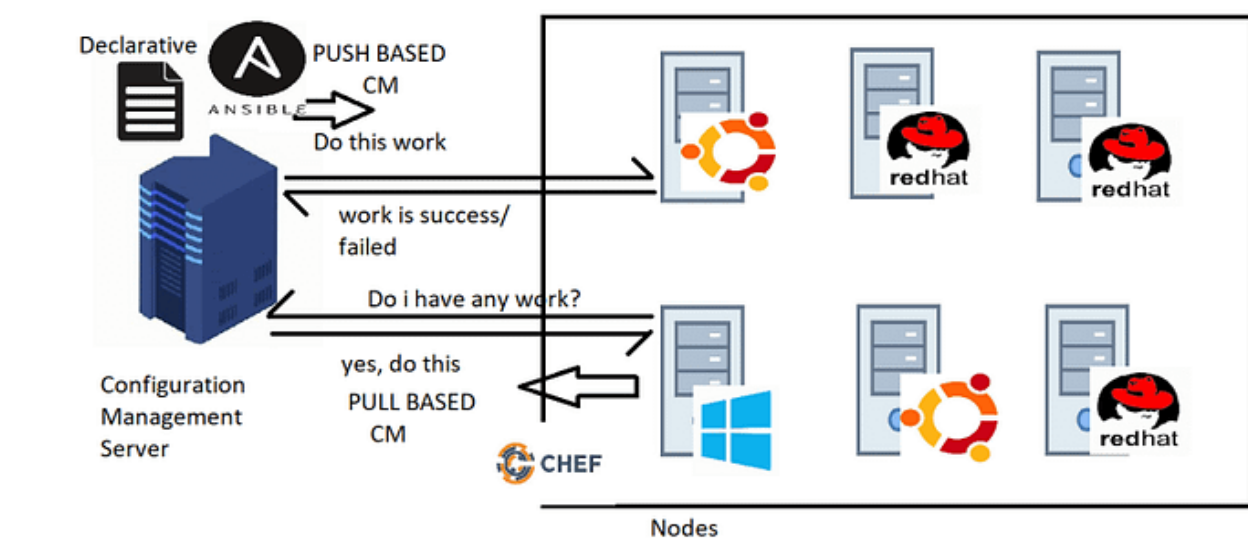
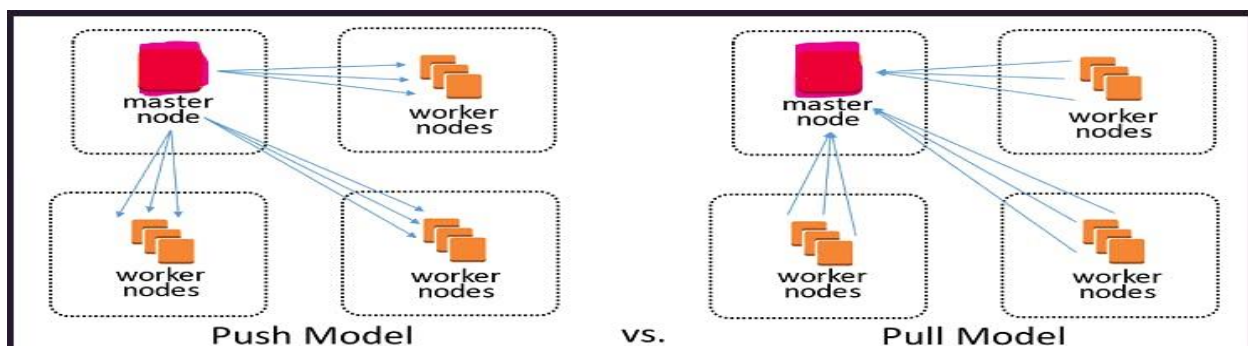
Ansible offers flexibility with both pull and push models. We must evaluate our operational needs to determine which model or combination thereof best aligns with our requirements for efficient configuration management and deployment.

Ansible Configuration Models:

1. Push Configuration Model: Ansible primarily utilizes a push model where the control node (Ansible server) initiates and pushes configuration changes and tasks to managed nodes. This process involves the control node establishing SSH connections (or WinRE for Windows) to managed nodes and executing tasks defined in playbooks. The push model ensures immediate deployment and execution of tasks, making it ideal for environments requiring real-time updates and rapid deployment cycles.

2. Pull Configuration Model (Ansible Pull): Alternatively, Ansible supports a pull model known as Ansible Pull. In this configuration, each managed node retrieves its configurations from a centralized source, typically a version control system or a configuration management server. Managed nodes use cron jobs or scheduled tasks to periodically pull configuration changes and apply them locally. This model is advantageous in environments where managed nodes have

restricted inbound network access or when nodes need to self-manage configuration updates independently.



***Note:** In our DevOps project, we leverage Ansible for efficient OS-level configuration management. Concurrently, Jenkins pipelines automate the deployment of our applications into designated environments, ensuring seamless integration and version tagging for streamlined development and release processes. This approach enhances reliability and consistency across our software delivery lifecycle, supporting agile practices and continuous improvement in our development workflows.

7. Terraform:

Terraform is an Infrastructure as Code (IaC) tool developed by HashiCorp that allows users to define and manage infrastructure configurations using a declarative language. It simplifies the

provisioning and management of cloud, on-premises, and hybrid infrastructure by treating infrastructure as code. Terraforms configuration files describe the desired state of the infrastructure, specifying resources such as virtual machines, networks, storage, and more.

Key features of Terraform include its ability to automate the deployment of complex infrastructure setups, manage dependencies between resources, and provide a consistent way to manage infrastructure across different providers like AWS, Azure, Google Cloud, and others. Terraform uses a plugin-based architecture that supports a wide range of providers, enabling users to manage heterogeneous environments seamlessly.

By using Terraform, organizations can achieve infrastructure agility, improve collaboration between teams through version-controlled configurations, and enforce best practices in infrastructure management, thereby accelerating application delivery and ensuring infrastructure reliability and consistency.

Terraform Architecture:

Terraform architecture mainly consists of the following components:

- Terraform Core
- Providers
- State file

Terraform Core

Terraforms core (also known as Terraform CLI) is built on a statically-compiled binary that's developed using the Go programming language.

This binary is what generates the command line tool (CLI) known as "terraform," which serves as the primary interface for Terraform users. It is open source and can be accessed on the Terraform GitHub repository.

Providers

Terraform providers are modules that enable Terraform to communicate with a diverse range of services and resources, including but not limited to cloud providers, databases, and DNS services.

Each provider is responsible for defining the resources that Terraform can manage within a particular service and translating Terraform configurations into API calls that are specific to that service.

Providers are available for numerous services and resources, including those developed by major cloud providers like AWS, Azure, and Google Cloud, as well as community-supported providers for various services. By utilizing providers, terraform users can maintain their infrastructure in a consistent and reproducible manner, regardless of the underlying service or provider.

State file

The Terraform State file is an essential aspect of Terraforms functionality. It is a JSON file that stores information about the resources that Terraform manages, as well as their current state and dependencies.

Terraform utilizes the state file to determine the changes that need to be made to the infrastructure when a new configuration is applied. It also ensures that resources are not unnecessarily recreated across multiple runs of Terraform.

The state file can be kept locally on the machine running Terraform or remotely using a remote backend like Azure Storage Account or Amazon S3, or HashiCorp Consul. It is crucial to safeguard the state file and maintain frequent backups since it contains sensitive information about the infrastructure being managed.

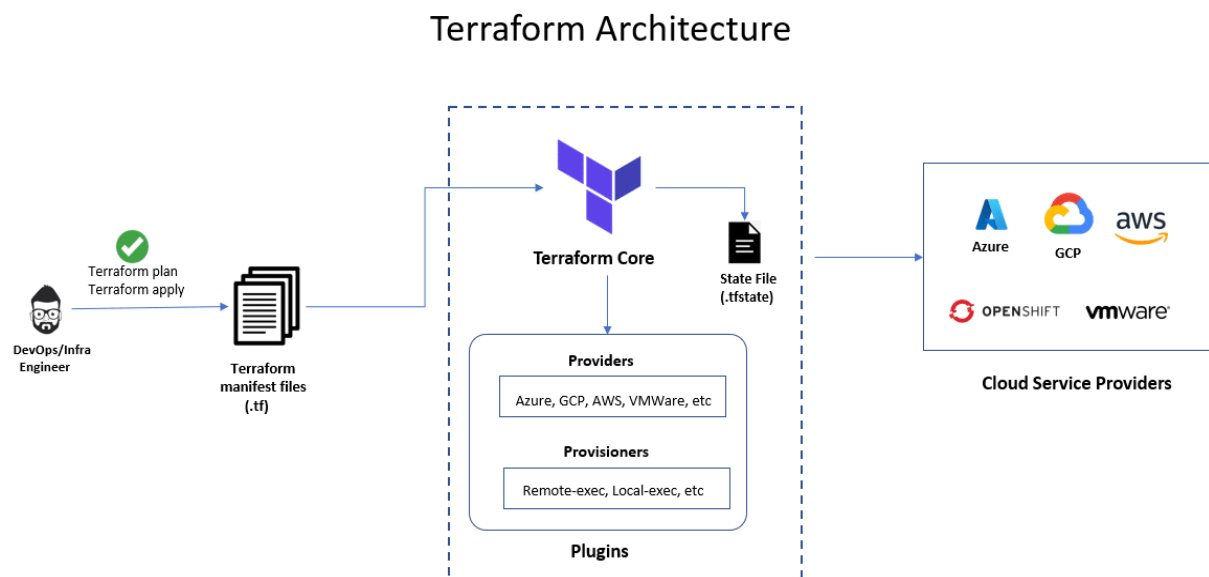


Figure 23

*Note: In my DevOps project we are using HashiCorp (HCL) Terraform for cloud Infrastructure configuration, we are defining infrastructure resources and configurations for a robust 3-tier e-commerce application environment on AWS. Here's a breakdown of what you're accomplishing:

1. Environment and Tags:

- `env`: Specifies the environment as "dev" and "prod".
- `tags`: Assigns metadata tags for resource identification and management, including company name, business unit, cost center, and project name.

2. VPC Configuration:

- `vpc`: Defines a Virtual Private Cloud (VPC) with subnets for different components: web, app, db, and public, each with specific CIDR blocks.

3. Default VPC and Route Table:

- `default_vpc_id` and `default_vpc_rt`: Reference default VPC and route table IDs.

4. Security and Networking:

- `allow_ssh_cidr` and `allow_prometheus_cidr`: Specify allowed CIDR ranges for SSH and Prometheus access.
- `zone_id`: Specifies the DNS zone ID.
- `kms_key_id` and `kms_key_arn`: Define the AWS KMS key for encryption.

5. Services Configuration:

- `rabbitmq`, `rds`, `document_db`, `elasticache`: Define configurations for RabbitMQ, RDS (Aurora MySQL), DocumentDB, and ElastiCache (Redis) instances with specified instance types and settings.

6. Load Balancers:

- `alb`: Configures both public and private Application Load Balancers with specified subnets and types.

7. Application Deployments:

- `apps`: Specifies configurations for different application components (cart, catalogue, user, shipping, payment, frontend) including instance types, desired capacity, load balancer references, and subnet references.

8. EKS Cluster:

- `eks`: Defines an Elastic Kubernetes Service (EKS) cluster with spot instances, specifying minimum and maximum sizes and instance types.

This configuration enables automated provisioning and management of a scalable, secure, and efficient infrastructure for an e-commerce application, utilizing various AWS services and components, all defined in a consistent and repeatable manner through Terraform.

8. Docker:

Docker is an open-source platform that automates the deployment, scaling, and management of applications using containerization. Containers package software and its dependencies into a standardized unit for software development, ensuring that applications run consistently across different computing environments. Docker's lightweight containers allow for quick application

deployment, efficient resource utilization, and enhanced scalability. It includes tools for creating, deploying, and managing containers, as well as Docker Hub, a repository for sharing container images. By isolating applications, Docker ensures that developers can build once and run anywhere, streamlining development workflows and improving DevOps practices.

Docker Architecture:

Docker's architecture comprises several key components that work together to enable containerization:

1. Docker Client:

- The Docker client is a command-line interface (CLI) that users interact with to manage Docker containers and images. Commands issued through the client are sent to the Docker daemon for execution.

2. Docker Daemon (docked):

- The Docker daemon runs on the host machine and manages Docker objects such as images, containers, networks, and volumes. It listens for Docker API requests and performs the requested actions.

3. Docker Engine:

- Docker Engine is the core part of Docker, comprising the Docker daemon, API, and CLI. It is responsible for building, running, and managing containers.

4. Docker Images:

- Docker images are read-only templates used to create containers. They contain the application and all dependencies needed to run it. Images are stored in Docker registries.

5. Docker Containers:

- Containers are lightweight, portable, and run isolated processes based on Docker images. They encapsulate an application and its dependencies, ensuring consistency across different environments.

6. Docker Registries:

- Registries, such as Docker Hub, are repositories where Docker images are stored and distributed. Users can pull images from public or private registries to create containers.

7. Docker Compose:

- Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure application services, allowing for easy management of complex environments.

Docker's architecture provides a reliable and streamlined environment for developing, deploying, and running applications across various platforms. By utilizing components such as the Docker

client, daemon, engine, images, containers, and registries, Docker ensures applications are packaged with all necessary dependencies, ensuring consistency, and eliminating environment-related issues. This architecture allows for rapid deployment, efficient resource utilization, and simplified management, enhancing scalability and flexibility. Docker's containerization technology isolates applications, making them portable across different environments, from development to production. This consistency and efficiency significantly improve DevOps workflows, enabling seamless integration, continuous delivery, and effective collaboration among development teams.

Note: In our DevOps project, we are not using standard Docker for application containerization. Instead, we are using AWS ECR (Elastic Container Registry) and integrating it with Jenkins to build desired artifacts with specific tags and store them in the designated repository.

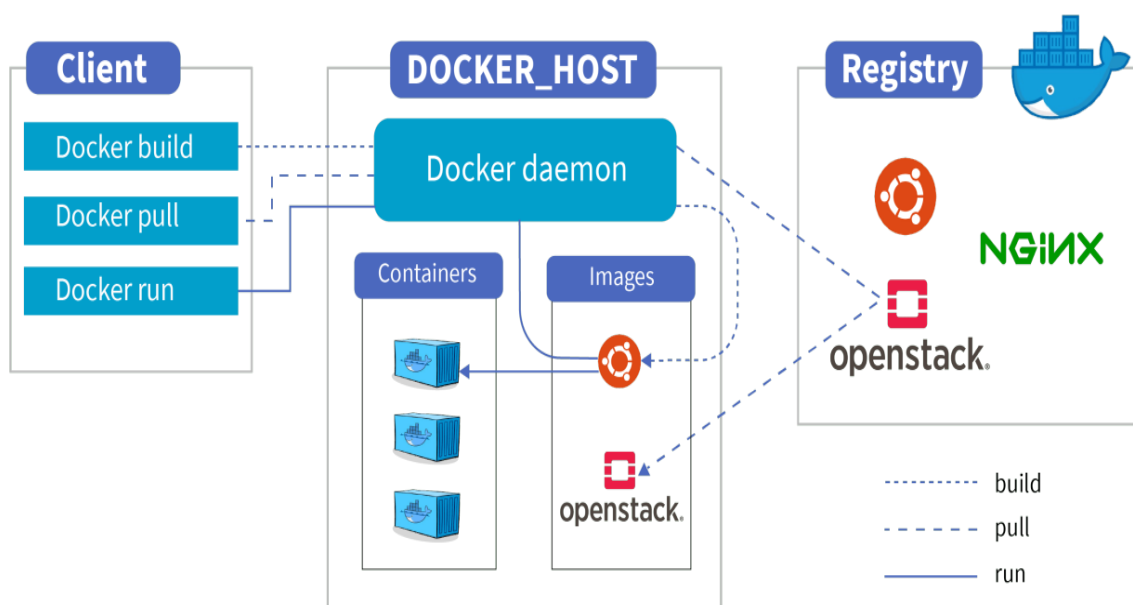


Figure 24

***Note:** In our DevOps project, we are not using standard Docker for application containerization. Instead, we are using AWS ECR (Elastic Container Registry) and integrating it with Jenkins to build desired docker images (artifacts) with specific tags and store them in the designated repository.

AWS ECR (Elastic Container Registry):

Amazon Elastic Container Registry (ECR) is a fully managed container image registry service by AWS, designed to store, manage, and deploy Docker container images securely. ECR integrates

seamlessly with Amazon ECS, EKS, and AWS Lambda, facilitating streamlined workflows for container-based applications. Key features include image vulnerability scanning, ensuring security and compliance, and fine-grained access control through AWS Identity and Access Management (IAM).

ECR supports automated replication of container images across multiple AWS regions, enhancing availability and reducing latency for global applications. It is designed to scale with your application, handling both small and large workloads effortlessly. Additionally, ECR integrates with continuous integration and continuous delivery (CI/CD) tools, enabling automated image building, testing, and deployment processes.

By using ECR, developers can ensure reliable, scalable, and secure storage for container images, optimizing their development and deployment pipelines and focusing more on building and deploying applications rather than managing infrastructure.

Drawbacks of Docker:

1. Complexity in Orchestration:

- Managing multiple containers can become complex without a robust orchestration tool like Kubernetes.

2. Security Concerns:

- Containers share the same OS kernel, leading to potential security vulnerabilities.

3. Storage Persistence:

- Handling persistent storage for stateful applications is challenging compared to traditional virtual machines.

4. Networking:

- Docker's networking model can be complicated, especially in multi-host setups.

5. Performance Overhead:

- Although minimal, there is still some overhead compared to running applications directly on the host OS.

Why Use AWS ECR (Elastic Container Registry):

Advantages of AWS ECR over Docker Hub:

1. Integration with AWS Ecosystem:

- ECR seamlessly integrates with other AWS services like ECS (Elastic Container Service), EKS (Elastic Kubernetes Service), IAM (Identity and Access Management), and CloudWatch.

2. Security and Compliance:

- ECR provides enhanced security features, such as image scanning for vulnerabilities, IAM roles for fine-grained access control, and encryption at rest with AWS KMS (Key Management Service).

3. Scalability and Reliability:

- AWS ECR offers high availability and automatically scales with your container image storage needs, ensuring reliability and performance without manual intervention.

4. Private Repository Management:

- ECR makes it easy to manage private repositories, with fine-grained permissions and policies.

5. Cost Efficiency:

- With ECR, you pay only for the storage you use and the data transferred out of the registry, potentially reducing costs compared to public registries.

6. Integration with CI/CD Pipelines:

- ECR integrates well with AWS Code Pipeline and other CI/CD tools, streamlining the build and deployment process.

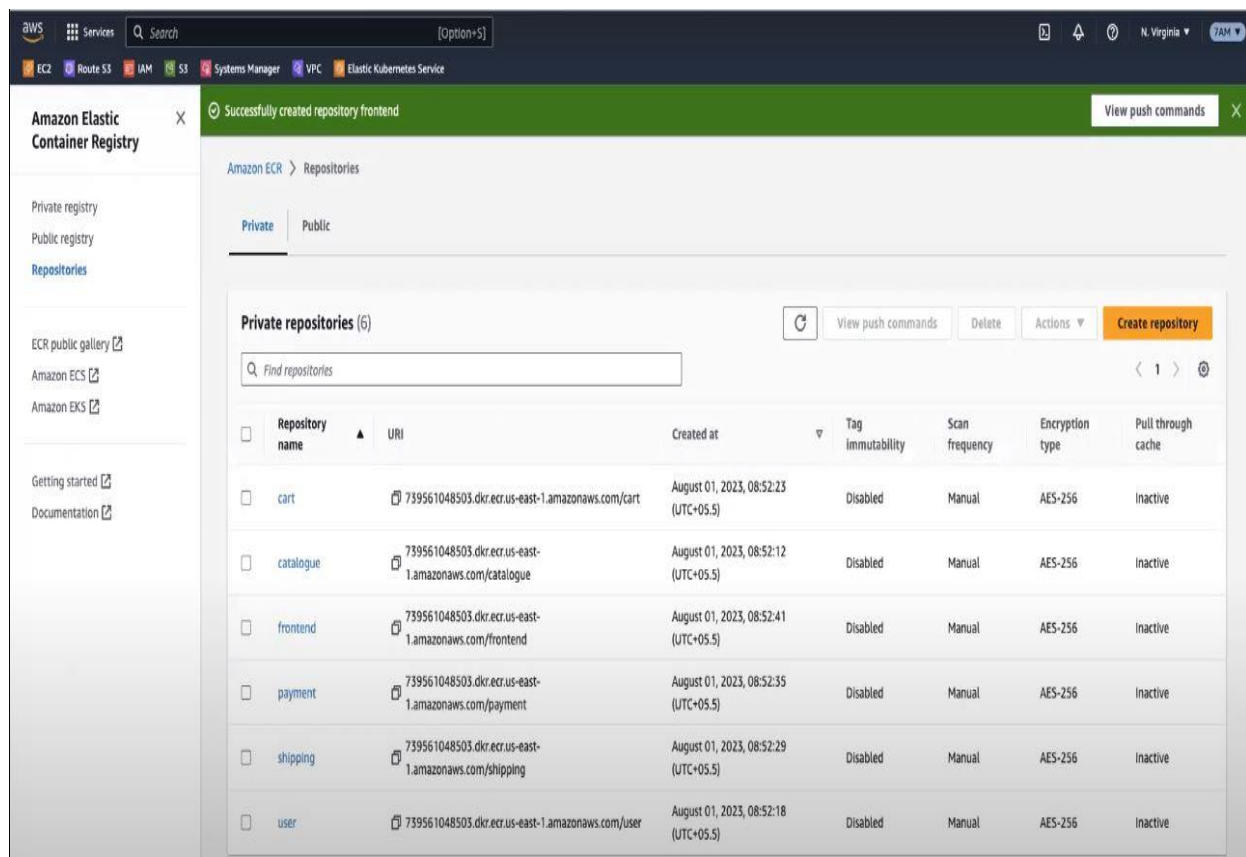


Figure 25

9. Kubernetes:

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes provides a robust framework for running distributed systems resiliently.

Key features include:

1. **Automated Deployment and Scaling:** Kubernetes can automatically deploy containers across a cluster and scale them up or down based on demand.
2. **Self-Healing:** It automatically replaces failed containers, restarts containers that fail, and kills containers that don't respond to user-defined health checks.
3. **Service Discovery and Load Balancing:** Kubernetes provides built-in service discovery and load balancing, ensuring seamless communication between containers.
4. **Storage Orchestration:** It allows automatic mounting of the storage system of your choice, whether from local storage, public cloud providers, or network storage solutions.
5. **Secret and Configuration Management:** Kubernetes manages sensitive information and configuration details securely.

Kubernetes enhances the flexibility, efficiency, and scalability of deploying and managing applications in dynamic and complex environments, making it a cornerstone of modern DevOps practices.

Kubernetes Architecture:

Kubernetes architecture consists of several key components that work together to manage containerized applications:

1. Master Node:

- **API Server:** Acts as the front-end for Kubernetes control plane. It validates and processes REST operations, serving as the primary management point.
- **Scheduler:** Assigns nodes for newly created pods based on resource availability and constraints.
- **Controller Manager:** Monitors cluster state and reconciles actual state with desired state, handling node and pod lifecycle events.
- **etcd:** Distributed key-value store storing all cluster data, ensuring consistency, and serving as Kubernetes' primary data store.

2. Worker Node:

- **Kubelet:** Agent running on each node, responsible for communication with the Kubernetes master and managing containers.
- **Container Runtime:** Software responsible for running containers, such as Docker or container.
- **Kube-proxy:** Manages network connectivity and maintains network rules, enabling communication between pods and external networks.

3. Pods:

- Smallest deployable units in Kubernetes, consisting of one or more containers sharing storage and network resources, managed as a single application.

4. Services:

- Kubernetes Service defines a logical set of Pods and a policy by which to access them, ensuring consistent networking and load balancing across the cluster.

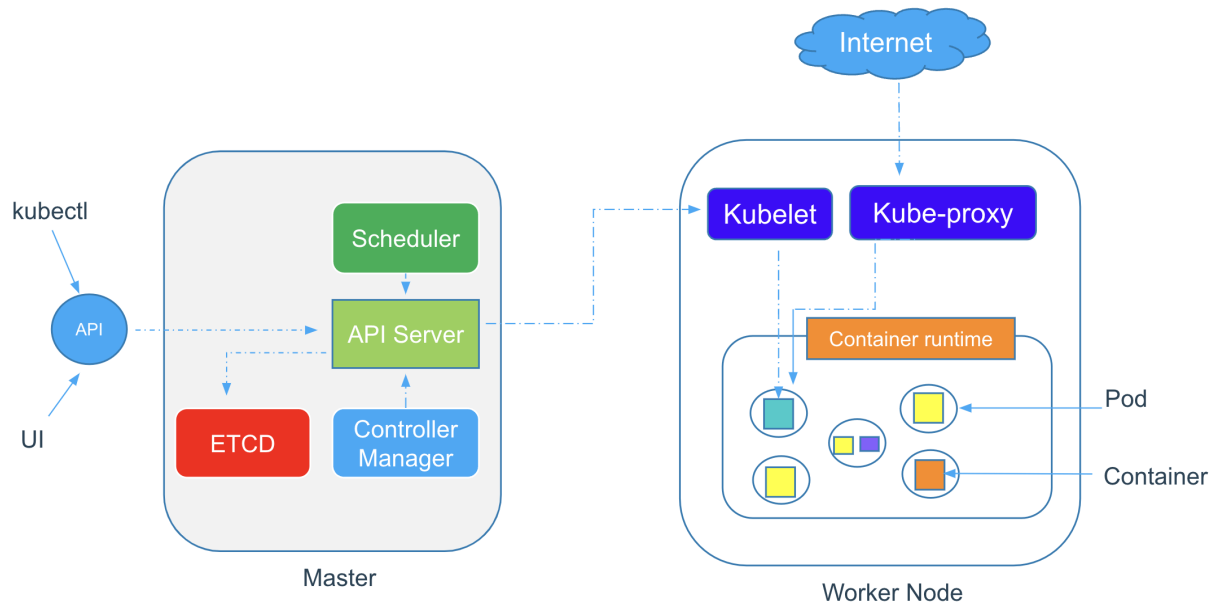


Figure 26

This architecture provides scalability, resilience, and automation for containerized applications, enabling efficient management and orchestration in complex environments.

AWS EKS (Elastic Kubernetes Service):

Amazon Elastic Kubernetes Service (EKS) is a managed service that simplifies running Kubernetes on AWS. EKS automates the deployment, management, and scaling of containerized applications using Kubernetes, eliminating the complexities of managing Kubernetes control plane infrastructure. It provides a highly available and secure Kubernetes control plane, distributed across multiple AWS availability zones for fault tolerance.

EKS integrates with various AWS services, including IAM for authentication, Elastic Load Balancing for load distribution, and Amazon VPC for networking. It supports Kubernetes tooling and plugins, enabling users to leverage the extensive Kubernetes ecosystem. With EKS, you can run standard Kubernetes without needing to modify your applications.

The service also supports both Windows and Linux workloads, allowing for flexibility in application deployment. EKS ensures that your Kubernetes environment is up-to-date with the latest security patches and features, providing a robust platform for building, deploying, and scaling modern applications with ease.

Helm:

Helm is a package manager for Kubernetes, designed to simplify the deployment and management of applications. It uses a packaging format called "charts," which are collections of files that describe a related set of Kubernetes resources. Helm charts can be used to define, install, and upgrade even the most complex Kubernetes applications.

Helm's key features include:

1. **Simplicity:** Helm reduces the complexity of Kubernetes application deployment by packaging resources together and managing them as a single entity.
2. **Reusability:** Charts can be reused across different projects, promoting consistency and best practices.
3. **Versioning:** Helm supports versioning of charts, allowing users to maintain and rollback versions of applications.
4. **Templating:** Helm charts can use templates, making them highly customizable for different environments.
5. **Dependency Management:** Helm can manage dependencies between charts, ensuring all required components are deployed together.

By using Helm, developers can automate application deployment, streamline updates, and maintain cleaner and more organized Kubernetes environments.

Helm Architecture:

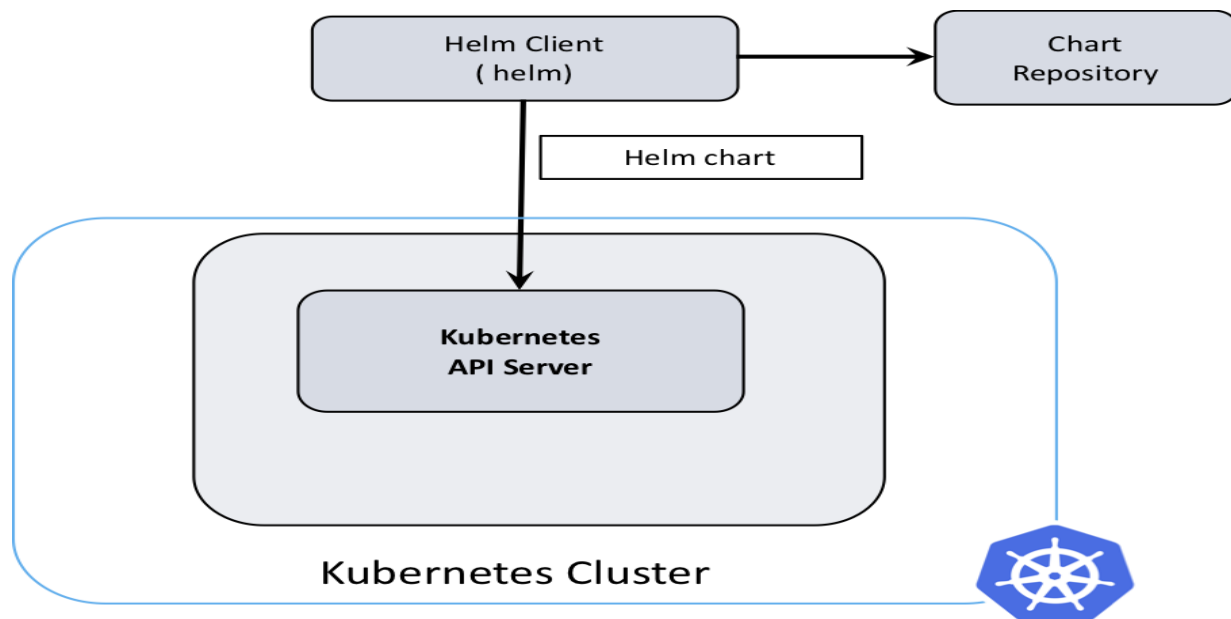


Figure 27

Ngix-Ingress-Controller:

The NGINX Ingress Controller is a crucial component in Kubernetes for managing inbound traffic to applications running on the cluster. It acts as an entry point, routing external requests to the appropriate services based on configurable rules and load balancing algorithms. This controller supports path-based routing, enabling multiple applications to share a single IP address and port combination efficiently.

Security features include SSL/TLS termination, IP whitelisting, and rate limiting, ensuring secure and controlled access to services. It integrates seamlessly with Kubernetes, leveraging annotations to customize and extend routing behavior without modifying the controller's core configuration. The NGINX Ingress Controller is highly scalable, capable of handling large volumes of traffic and scaling alongside Kubernetes clusters.

Overall, it simplifies the management of traffic flow into the Kubernetes environment, enhances application availability, and provides robust capabilities for managing ingress traffic efficiently and securely.

*Note: In our DevOps project, we utilize AWS ECR with private repositories to store our Docker images as artifacts with specified tags. For deploying our API services into AWS EKS, we use Helm chart templating and deployment services. This setup also includes Horizontal Pod Autoscaling (HPA) managed through Helm charts.

We leverage Kubernetes Services and Service Accounts for managing network access and permissions. Authentication is handled via OIDC. Additionally, we provision Application Load Balancers (ALB) and the Ingress controller using Terraform.

This comprehensive approach ensures our application is efficiently deployed, scaled, and managed within the EKS cluster, with secure access and automated infrastructure provisioning.

Deploying our services into the EKS cluster using ECR images:

1. Creating a **Dockerfile**: We begin by crafting a Dockerfile for our application.
2. Pushing to **Amazon ECR**: The Dockerfile is then pushed to Amazon ECR (Elastic Container Registry).
3. Helm **Deployment**: Kubernetes deployment is handled via Helm, offering flexibility to update software versions with each new release.
4. Service **Creation**: Services are established to either expose internal components to other services or make certain services accessible externally.
5. Service **Account**: To enable secure communication, a service account is created, providing the necessary identity.
6. Namespace **Usage**: Throughout these processes, we utilize the default namespace for organizational purposes.

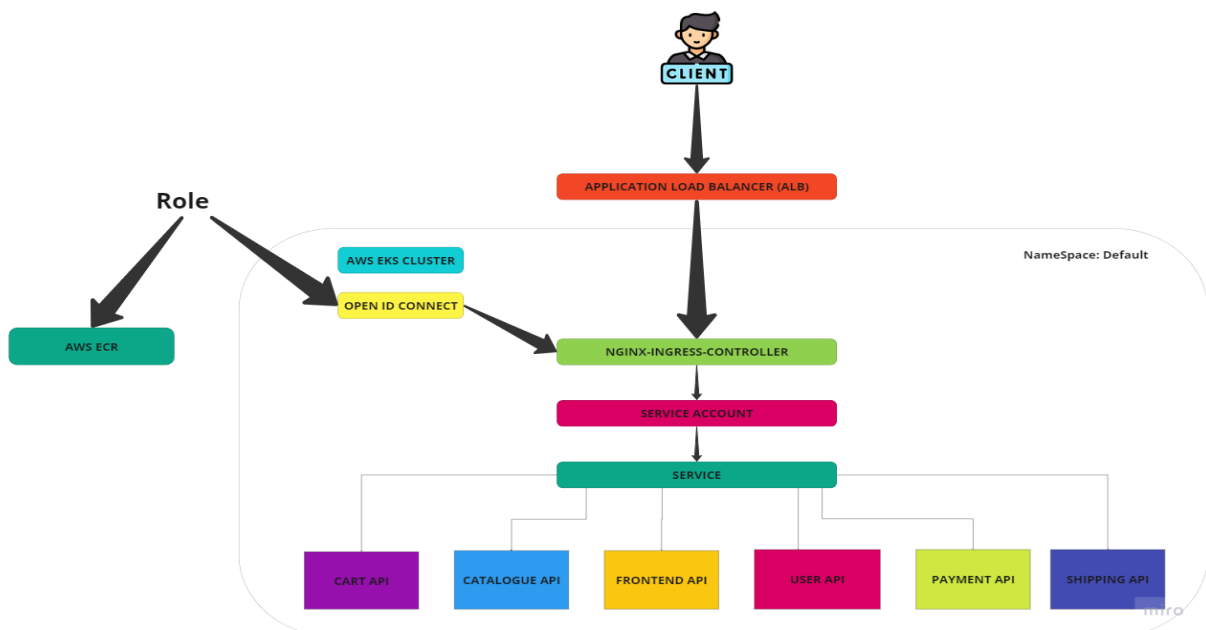


Figure 28

***Note:** In this DevOps project, we utilize the aws-parameter-init-container to fetch parameters and secrets from the AWS Parameter Store. For schema loading in the catalogue and user API services, we employ the schema-init-container. During the service deployment in EKS cluster, these containers act as sidecars, ensuring that the necessary parameters are fetched and the services remain operational.

Horizontal Pod Autoscaling (HPA):

Horizontal Pod Autoscaling (HPA) in Amazon EKS (Elastic Kubernetes Service) automatically adjusts the number of pods in a deployment, replica set, or stateful set based on CPU utilization or other select metrics. HPA helps maintain application performance and resource efficiency by scaling pods up or down in response to workload changes. Configuration involves setting target metrics and thresholds, allowing Kubernetes to monitor and adjust the pod count accordingly. This dynamic scaling ensures that applications can handle varying loads while optimizing resource usage, leading to better cost management and consistent performance in the EKS environment.

HPA is well-suited for applications with fluctuating traffic, ensuring reliability and performance.

1. **Scalability:** HPA adjusts the number of pods based on workload, providing better scalability for handling varying traffic loads.
2. **High Availability:** By distributing the load across multiple pods, HPA ensures higher availability and fault tolerance.
3. **Resource Efficiency:** HPA optimizes resource usage by adding or removing pods as needed, avoiding resource overprovisioning, and reducing costs.
4. **Performance:** It maintains consistent application performance during traffic spikes by rapidly scaling out the number of pods.

K9'S Tool:

K9s is a highly efficient, terminal-based UI tool designed to simplify the management of Kubernetes clusters. It offers a streamlined interface that allows users to interact with Kubernetes resources directly from the command line. With K9s, users can navigate through various Kubernetes objects, including pods, nodes, services, deployments, and namespaces, with ease.

One of the key features of K9s is its ability to provide real-time monitoring and insights into the state of the cluster. It continuously tracks and displays the health and status of the cluster components, enabling users to quickly identify and address issues. Additionally, K9s offers comprehensive logging capabilities, allowing users to view and analyze logs from individual pods and containers, facilitating effective troubleshooting and debugging.

K9s also supports a wide range of operations, such as scaling deployments, executing commands within pods, port-forwarding, and deleting resources. These functionalities make it a powerful tool for both administrators and developers, enhancing their productivity by simplifying complex Kubernetes tasks.

```
3.92.240.139 - PuTTY
Context: arn:aws:eks:us-east-1:828448425071:cluster/prod-eks
Cluster: arn:aws:eks:us-east-1:828448425071:cluster/prod-eks
User: arn:aws:eks:us-east-1:828448425071:cluster/prod-eks
K8s Rev: v0.31.8 <v0.32.4
K8s Rev: v1.29.3-eks-ade7111
CPU: 0%
MEM: 1%

Pod(s) (all) [10]
NAMESPACE NAME PF READY STATUS RESTARTS CPU MEM %CPU/R %CPU/L %MEM/R %MEM/L IP NODE
default ingress-nginx-controller-68fdd96f7b-p4pzt 1/1 Running 0 2 53 2 n/a n/a n/a 10.10.3.174 ip-10-10-3-177.ec2.inte
kube-system aws-node-8j1df 2/2 Running 0 4 42 8 n/a n/a n/a 10.10.3.177 ip-10-10-3-177.ec2.inte
kube-system aws-node-c5np8 2/2 Running 0 5 58 10 n/a n/a n/a 10.10.2.198 ip-10-10-2-198.ec2.inte
kube-system aws-node-rnnhh 2/2 Running 0 5 58 10 n/a n/a n/a 10.10.3.59 ip-10-10-3-59.ec2.inte
kube-system coredns-54d6f577c6-w5hqk 1/1 Running 0 2 12 2 n/a 18 7 10.10.2.182 ip-10-10-2-198.ec2.inte
kube-system coredns-54d6f577c6-z67pq 1/1 Running 0 2 13 2 n/a 18 7 10.10.3.52 ip-10-10-3-59.ec2.inte
kube-system kube-proxy-2pgfg 1/1 Running 0 1 13 1 n/a n/a n/a 10.10.3.59 ip-10-10-3-59.ec2.inte
kube-system kube-proxy-qft8d 1/1 Running 0 1 12 1 n/a n/a n/a 10.10.2.198 ip-10-10-2-198.ec2.inte
kube-system kube-proxy-qgr9n 1/1 Running 0 1 13 1 n/a n/a n/a 10.10.3.177 ip-10-10-3-177.ec2.inte
kube-system metrics-server-6d94bc8694-7srtc 1/1 Running 0 4 19 4 n/a 9 n/a 10.10.3.170 ip-10-10-3-59.ec2.inte
```

Figure 29

```
3.92.240.139 - PuTTY
Context: arn:aws:eks:us-east-1:828448425071:cluster/prod-eks
Cluster: arn:aws:eks:us-east-1:828448425071:cluster/prod-eks
User: arn:aws:eks:us-east-1:828448425071:cluster/prod-eks
K8s Rev: v0.31.8 <v0.32.4
K8s Rev: v1.29.3-eks-ade7111
CPU: 1%
MEM: 5%

Pod(s) (all) [25]
NAMESPACE NAME PF READY STATUS RESTARTS CPU MEM %CPU/R %CPU/L %MEM/R %MEM/L IP NODE
default cart-64f579b8d-2d1mw 1/1 Running 0 1 37 0 0 7 5 10.10.3.119 ip-10-10-3-177.ec2.inte
default cart-64f579b8d-mq49b 1/1 Running 0 1 37 0 0 7 5 10.10.2.250 ip-10-10-2-198.ec2.inte
default catalogue-6bbd9f766b-f4dp6 1/1 Running 0 1 44 0 0 8 6 10.10.3.224 ip-10-10-3-59.ec2.inte
default catalogue-6bbd9f766b-txwtf 1/1 Running 0 1 44 0 0 8 6 10.10.2.4 ip-10-10-2-198.ec2.inte
default filebeat-filebeat-96ndn 1/1 Running 0 14 64 14 1 64 32 10.10.3.176 ip-10-10-3-177.ec2.inte
default filebeat-filebeat-sng6h 1/1 Running 0 29 63 28 2 63 31 10.10.2.243 ip-10-10-2-198.ec2.inte
default filebeat-filebeat-z48cn 1/1 Running 0 28 61 28 2 61 30 10.10.3.54 ip-10-10-3-59.ec2.inte
default frontend-c6b9cb77f-khdsx 1/1 Running 0 1 8 0 0 1 1 10.10.3.115 ip-10-10-3-177.ec2.inte
default frontend-c6b9cb77f-rp8zb 1/1 Running 0 1 8 0 0 1 1 10.10.2.200 ip-10-10-2-198.ec2.inte
default ingress-nginx-controller-68fdd96f7b-p4pzt 1/1 Running 0 3 114 3 n/a 127 n/a 10.10.3.174 ip-10-10-3-177.ec2.inte
default payment-567b69bf5-klp99 1/1 Running 0 1 42 0 0 4 2 10.10.2.95 ip-10-10-2-198.ec2.inte
default payment-567b69bf5-q288d 1/1 Running 0 1 42 0 0 4 2 10.10.3.91 ip-10-10-3-59.ec2.inte
default shipping-bbc46c798-98r2f 1/1 Running 0 2 1000 0 0 48 39 10.10.3.166 ip-10-10-3-59.ec2.inte
default shipping-bbc46c798-cd5vb 1/1 Running 0 2 636 0 0 31 24 10.10.2.94 ip-10-10-2-198.ec2.inte
default user-55d66bcbfb-b9d8v 1/1 Running 0 1 43 0 0 8 6 10.10.3.7 ip-10-10-3-177.ec2.inte
default user-55d66bcbfb-qpf5 1/1 Running 0 1 43 0 0 8 6 10.10.2.236 ip-10-10-2-198.ec2.inte
kube-system aws-node-8j1df 2/2 Running 0 5 60 10 n/a n/a n/a 10.10.3.177 ip-10-10-3-177.ec2.inte
kube-system aws-node-c5np8 2/2 Running 0 3 60 6 n/a n/a n/a 10.10.2.198 ip-10-10-2-198.ec2.inte
kube-system aws-node-rnnhh 2/2 Running 0 4 60 8 n/a n/a n/a 10.10.3.59 ip-10-10-3-59.ec2.inte
kube-system coredns-54d6f577c6-w5hqk 1/1 Running 0 2 15 2 n/a 22 9 10.10.2.182 ip-10-10-2-198.ec2.inte
kube-system coredns-54d6f577c6-z67pq 1/1 Running 0 2 15 2 n/a 22 9 10.10.3.52 ip-10-10-3-59.ec2.inte
kube-system kube-proxy-2pgfg 1/1 Running 0 1 13 1 n/a n/a n/a 10.10.3.59 ip-10-10-3-59.ec2.inte
kube-system kube-proxy-qft8d 1/1 Running 0 1 13 1 n/a n/a n/a 10.10.2.198 ip-10-10-2-198.ec2.inte
kube-system kube-proxy-qgr9n 1/1 Running 0 1 13 1 n/a n/a n/a 10.10.3.177 ip-10-10-3-177.ec2.inte
kube-system metrics-server-6d94bc8694-vmtdz 1/1 Running 0 5 23 5 n/a 11 n/a 10.10.3.137 ip-10-10-3-177.ec2.inte
```

Figure 30

Note: K9s screenshots display all API services, the NGINX Ingress Controller, Kubernetes API services, AWS services, and Kubernetes metric services in their running state, ensuring the entire application is up and operational.

10. ELK Stack (Elasticsearch, Logstash, Kibana):

The ELK Stack, comprised of Elasticsearch, Logstash, and Kibana, is a powerful open-source suite for searching, analyzing, and visualizing log and event data in real time.

- 1. Elasticsearch:** At the core of the stack, Elasticsearch is a distributed search and analytics engine. It enables users to store, search, and analyze large volumes of data quickly and in near real-time. Elasticsearch is known for its scalability, high availability, and ability to handle diverse data types.
- 2. Logstash:** This is a server-side data processing pipeline that ingests data from various sources simultaneously, transforms it, and sends it to a designated "stash" like Elasticsearch. Logstash can handle complex data transformations and enrichments, making it a versatile tool for collecting and processing log data, metrics, and other types of events.
- 3. Kibana:** Serving as the frontend interface, Kibana allows users to visualize data stored in Elasticsearch. It provides a rich set of visualization options, including histograms, line graphs, pie charts, and maps. Kibana is highly interactive, allowing users to create dynamic dashboards, explore data in real-time, and share insights.

Together, the ELK Stack is widely used for log management and analysis, security monitoring, and operational intelligence. It helps organizations gain deep insights into their data, enhance system performance, and troubleshoot issues more effectively. The seamless integration and flexibility of the ELK Stack make it a popular choice for managing and visualizing big data.

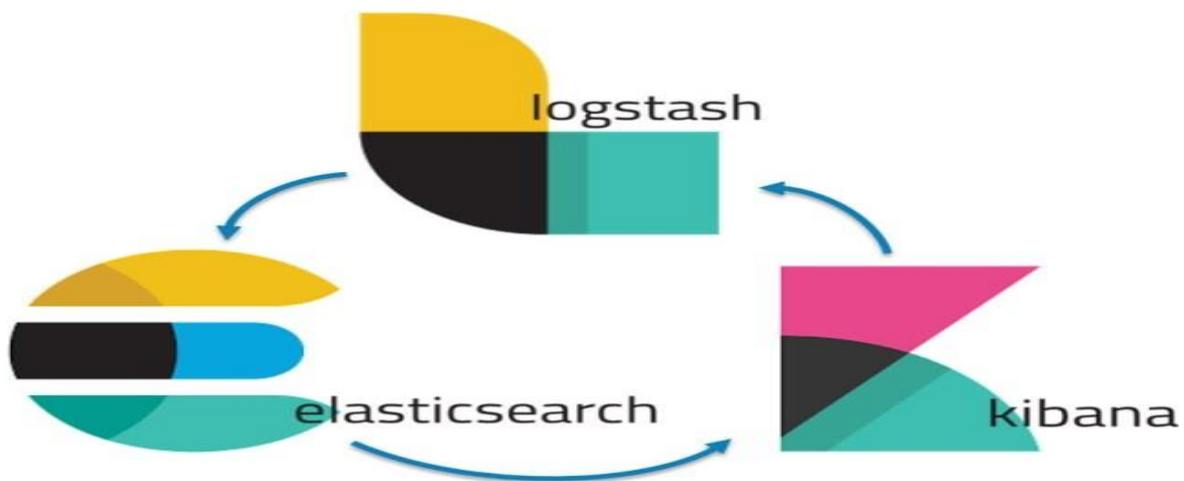


Figure 31

ELK Stack Workflow:

The ELK Stack, comprising Elasticsearch, Logstash, and Kibana, along with Filebeat, provides a comprehensive solution for log and event data collection, processing, storage, and visualization.

- 1. Filebeat:** Filebeat is a lightweight log shipper that collects and forwards log data to Logstash or Elasticsearch. It efficiently monitors log files, capturing data in real-time with minimal resource consumption, ensuring effective log transportation from various sources like system logs and application logs.
- 2. Logstash:** Once Filebeat ships the data, Logstash processes it by applying filters for transformation and enrichment. It can handle complex data processing tasks, ingesting data from multiple sources, and preparing it for indexing by Elasticsearch.
- 3. Elasticsearch:** Elasticsearch indexes and stores the processed data, making it searchable and analyzable in near real-time. Its distributed architecture ensures scalability and fault tolerance, allowing it to handle large volumes of data efficiently.
- 4. Kibana:** Kibana serves as the visualization layer, providing an intuitive web interface for exploring and visualizing data stored in Elasticsearch. Users can create dynamic dashboards, generate various visualizations like charts and graphs, and perform detailed data analysis, gaining actionable insights.

Enhanced Workflow with Filebeat

- **Data Collection:** Filebeat captures log data from diverse sources.
- **Data Processing:** Logstash enriches and processes this data.
- **Data Indexing:** Elasticsearch indexes the processed data for efficient querying.
- **Data Visualization:** Kibana visualizes the data, enabling interactive dashboards and insights.

Incorporating Filebeat into the ELK Stack enhances data ingestion capabilities, making log management and analysis more efficient and streamlined.

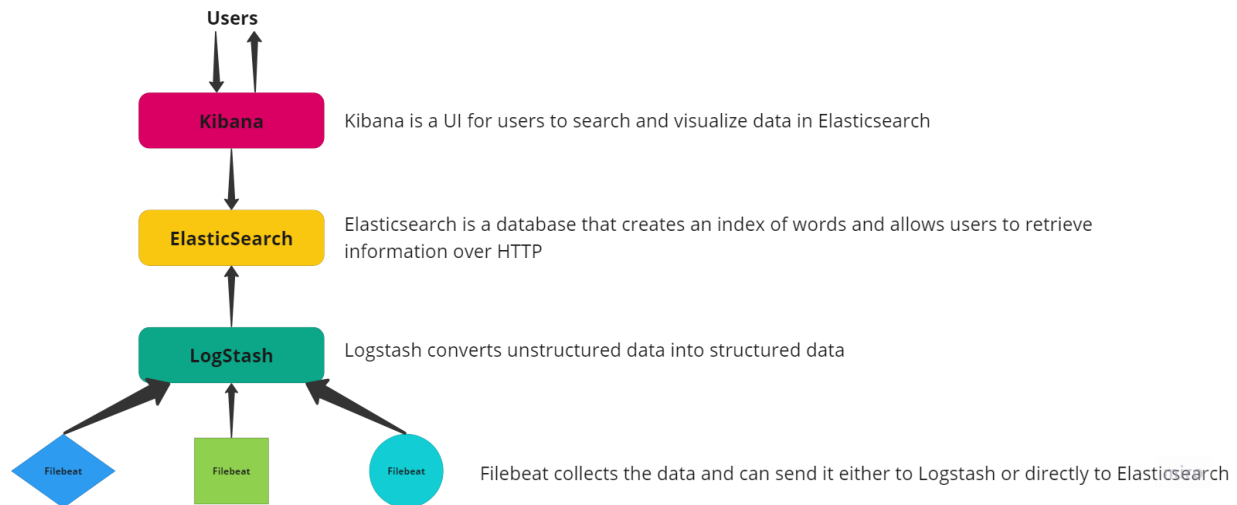


Figure 32

Project Approach using ELK Stack for Log Collection and Management:

In our DevOps project, integrating ELK for log collection is aimed at optimizing data management. Logs from the Payment, Frontend and Shipping are processed through Logstash for structured formatting, while logs from the Cart, Catalogue and User are directly indexed in Elasticsearch. This approach ensures efficient data organization, supports real-time monitoring, and enables detailed analysis. By utilizing Elasticsearch's powerful indexing capabilities and Logstash's data transformation features, we centralize log management, streamline troubleshooting processes, and gain valuable insights into application performance and user behavior. This implementation enhances operational efficiency and helps maintain robust database management practices throughout the project lifecycle.

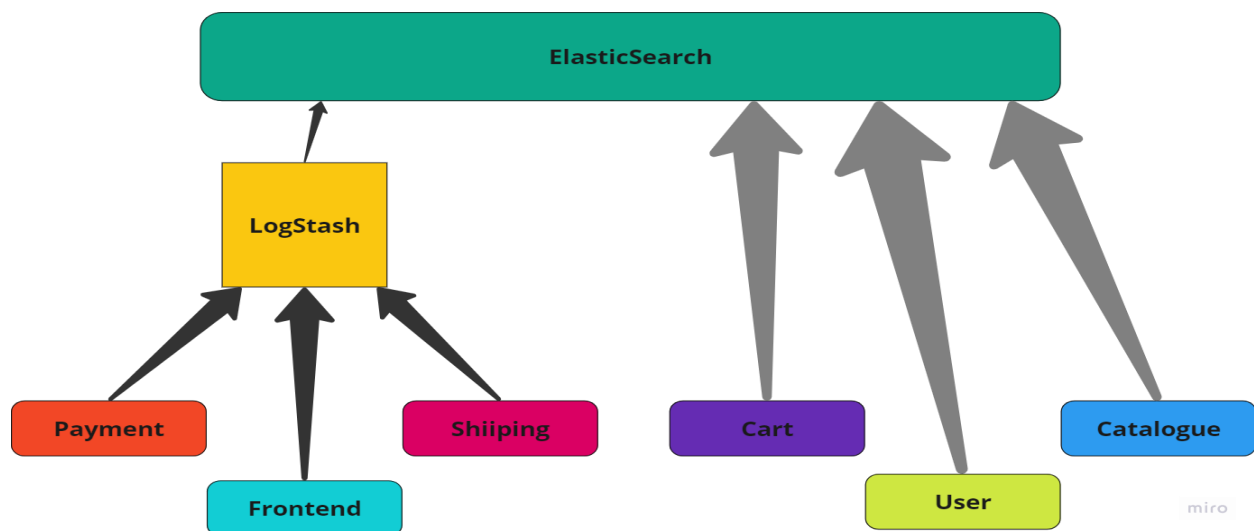


Figure 33

Four Golden Signals from Google's Site Reliability Engineering (SRE):

The Four Golden Signals, as defined by Google's Site Reliability Engineering (SRE) practices, are essential metrics for monitoring system performance and reliability:

1. **Latency:** This measures the time taken to service a request. It includes two aspects: the time for successful requests and the time for failed requests. Latency is crucial for understanding user experience, as higher latency can indicate performance bottlenecks.
2. **Traffic:** This metric indicates the load on the system, typically measured in requests per second or data volume per second. Monitoring traffic helps in capacity planning and ensures that the system can handle current and future load.
3. **Errors:** This signal tracks the rate of failed requests or operations. Errors can indicate underlying issues in the system, such as bugs, configuration problems, or resource exhaustion. Monitoring errors helps in maintaining system reliability.
4. **Saturation:** Saturation measures how "full" the system is, often reflecting resource utilization like CPU, memory, or disk usage. High saturation levels can lead to performance degradation and increased latency, making it crucial to monitor and manage.

Together, these signals provide a comprehensive view of system health, aiding in proactive issue detection and resolution.

Note: In our DevOps project, we are focused on monitoring logs, specifically Nginx logs, rather than business logs. We aim to collect valuable and relevant information from Nginx logs while avoiding irrelevant data.

We use the ELK Stack to process logs related to latency, traffic, and errors. For saturation-related logs, we utilize Prometheus and Grafana. The ELK Stack enables us to efficiently analyze and visualize log data for performance and error monitoring, while Prometheus and Grafana provide real-time metrics and dashboards to monitor resource utilization and system saturation, ensuring comprehensive monitoring coverage.

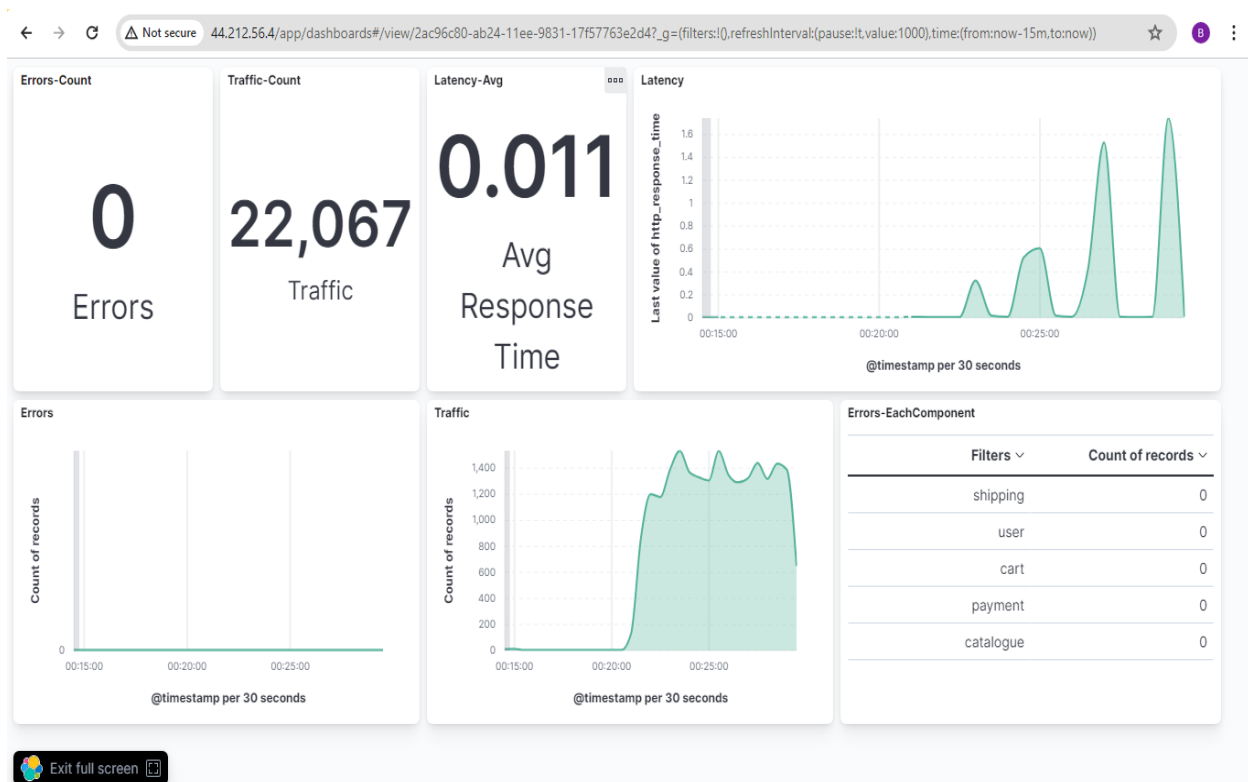


Figure 34

Note: We can get a clear graph of these three signals (Latency, Traffic, Errors) by generating continuous data. We can achieve this by using an automated script called “roboshop-load-gen,” which continuously generates load to produce the necessary data.

11. Prometheus and Grafana:

Prometheus:

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. Developed originally by SoundCloud in 2012 and now part of the Cloud Native Computing Foundation (CNCF), Prometheus is widely used for collecting and storing metrics as time series data, recording information with timestamps, labels, and optional key-value pairs.

Key Features:

- **Data Collection:** Prometheus scrapes metrics from instrumented jobs, servers, and applications using a pull-based model. It supports over 10,000 samples per second.
- **Data Storage:** Metrics are stored as time series data, efficiently stored on disk, enabling powerful querying capabilities.

- **Query Language:** Prometheus provides PromQL (Prometheus Query Language), which allows for flexible and sophisticated queries on the stored data, essential for creating dashboards and alerts.
- **Alerting:** Integrated with Alertmanager, Prometheus can generate alerts based on user-defined rules and send notifications through various channels like email, Slack, or custom webhooks.
- **Visualization:** While Prometheus includes a basic web UI, it is often paired with Grafana for advanced visualization and dashboarding capabilities.
- **Service Discovery:** Prometheus can dynamically discover and monitor services using various mechanisms, such as Kubernetes API, Consul, Eureka, and cloud provider APIs. This ensures that Prometheus can adapt to changes in the infrastructure without manual intervention.

Usage:

Prometheus is ideal for monitoring dynamic cloud-native environments, particularly Kubernetes. It excels at providing real-time metrics, supporting microservices architectures, and offering robust integration with various exporters and client libraries for different programming languages. Its modular and extensible nature allows it to be tailored to specific monitoring needs, making it a critical component in modern DevOps and SRE practices. By leveraging service discovery, Prometheus ensures that monitoring is robust, dynamic, and scalable, aligning well with modern infrastructure practices.

Prometheus Architecture:

Prometheus follows a straightforward yet powerful architecture designed for monitoring modern cloud-native environments:

Prometheus operates using a server-agent model. The core components include:

1. **Prometheus Server:** Responsible for scraping and storing time series data based on configured scraping intervals. It uses a pull-based approach to gather metrics from instrumented targets.
2. **Metrics:** Data collected from various jobs, servers, and applications are stored as time series in a local time-series database.
3. **Alert manager:** Handles alerts sent by Prometheus servers based on predefined rules. It manages the grouping, deduplication, and routing of alerts to different notification integrations like email or Slack.

4. **Visualization:** While Prometheus has a basic built-in UI for querying and visualizing metrics, it is commonly paired with Grafana for advanced visualization and dashboarding capabilities.
5. **Service Discovery:** Supports multiple service discovery mechanisms (e.g., Kubernetes, Consul) for dynamically discovering and monitoring targets. Prometheus' architecture ensures flexibility, scalability, and reliability in monitoring distributed systems, making it a popular choice in DevOps and SRE practices.

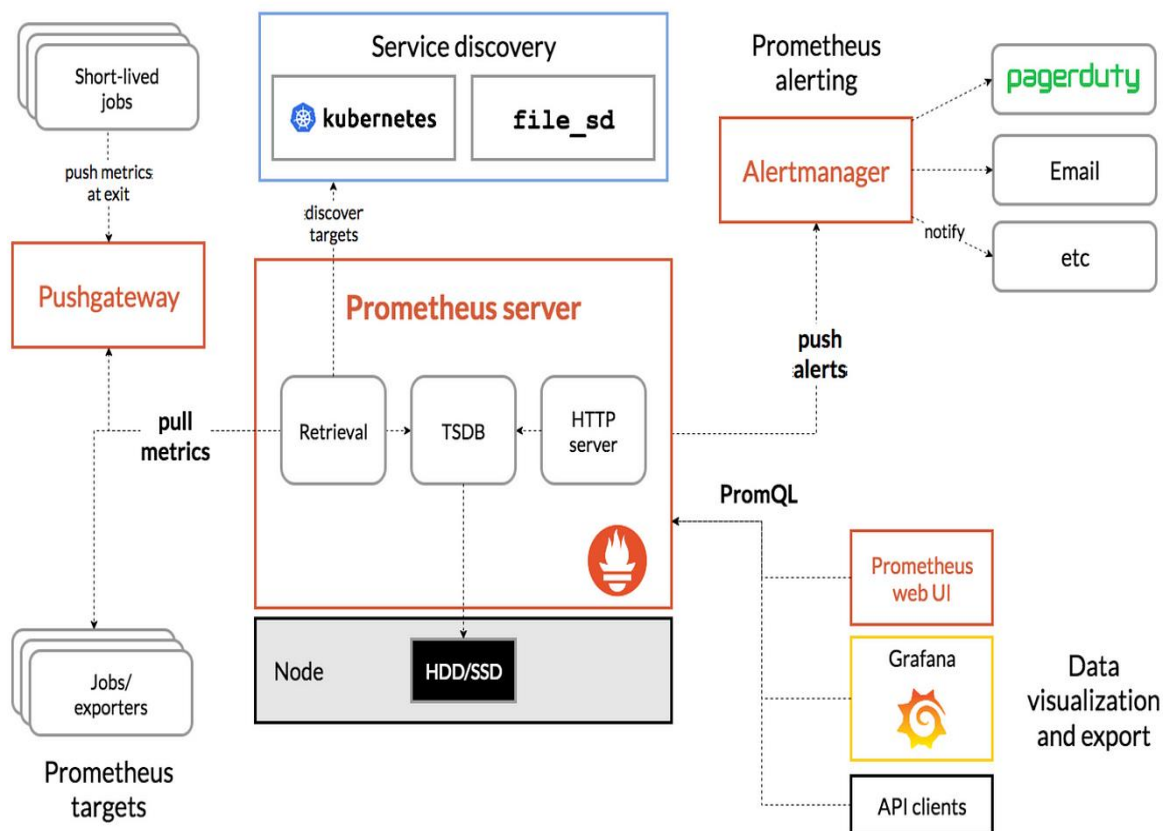


Figure 35

Prometheus Node-Exporter:

Prometheus Node Exporter is a specialized Prometheus exporter designed for Linux and Unix systems. It gathers vital system-level metrics including CPU usage, memory usage, disk I/O stats, and network traffic data. These metrics are essential for monitoring the health and performance of servers within a distributed environment. By exposing metrics via HTTP on a default port (usually 9100), the Node Exporter enables Prometheus to scrape and store time-series data, facilitating real-time monitoring, alerting, and analysis. This integration is fundamental for

ensuring optimal operational efficiency and preemptive issue resolution in modern infrastructure and cloud-native deployments.

Prometheus and ELK Integration:

Integrating Prometheus with ELK (Elasticsearch, Logstash, Kibana) involves configuring Prometheus to push metrics data to Elasticsearch using a compatible node-exporter or plugin. This allows Prometheus to store time-series data in Elasticsearch, which can then be visualized and analyzed alongside logs ingested by Logstash and displayed using Kibana. This integration provides a comprehensive monitoring solution, combining metrics-based monitoring from Prometheus with log-based insights from ELK for enhanced operational visibility and troubleshooting capabilities.

*Note: Prometheus Node-Exporter will export the Nginx Log metrics to Prometheus and these are Transferred to ELK Logstash.

Service Discovery:

In our project, Service Discovery is crucial for automating the detection and monitoring of remote server IP addresses. This mechanism enables Prometheus to dynamically identify instances, nodes, or services within our system or network, eliminating the need for manual updates to server IP configurations. By leveraging Service Discovery tools such as Kubernetes or Consul, Prometheus configures itself to automatically scrape metrics from newly discovered targets.

For example, Kubernetes' service discovery capabilities allow Prometheus to continuously monitor pods and services based on defined labels and selectors. This approach ensures that Prometheus adapts seamlessly to changes in infrastructure, scaling, or reconfiguration without intervention.

Implementing Service Discovery with Prometheus not only simplifies operational management but also enhances monitoring effectiveness and responsiveness. It supports the scalability and reliability required in modern cloud-native and distributed system architectures by ensuring that all relevant targets are consistently monitored for performance metrics. This automation facilitates proactive monitoring and alerting, enabling timely responses to issues and maintaining the overall health and availability of the monitored infrastructure.

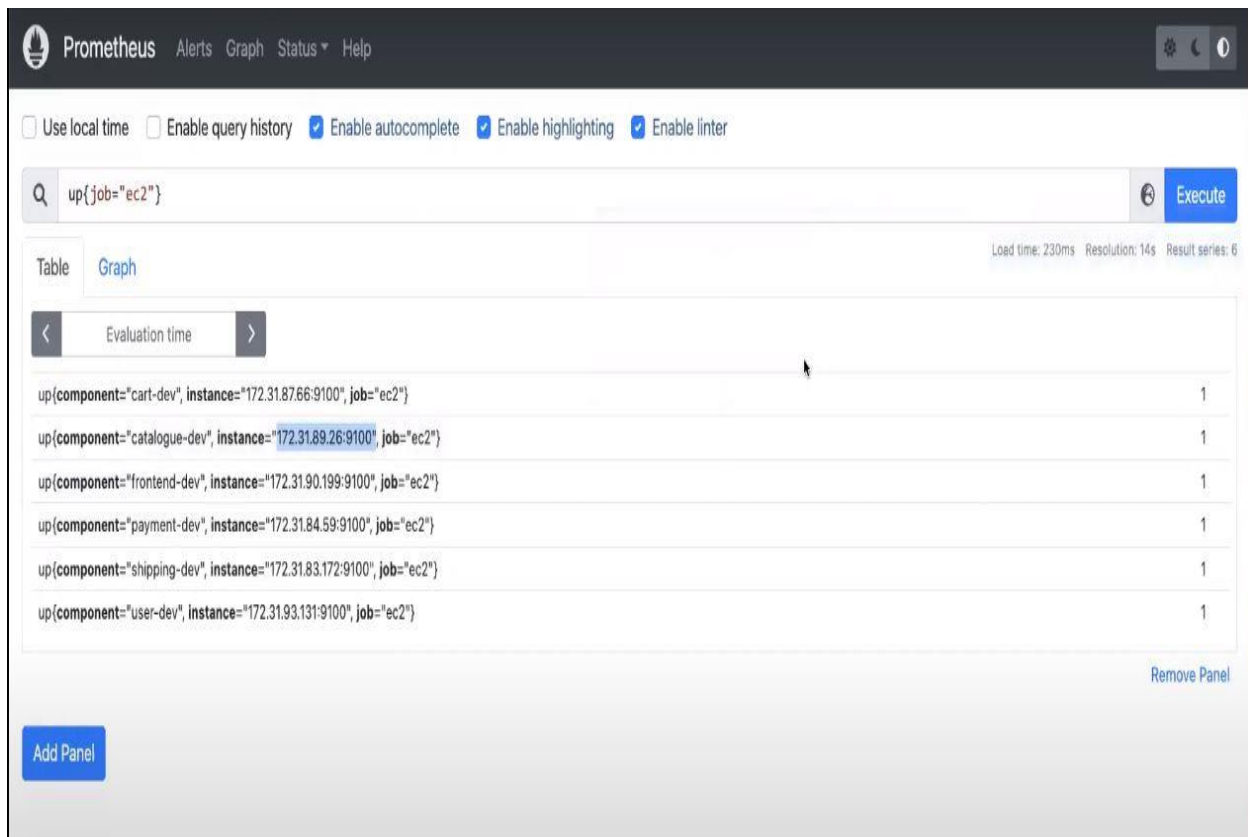


Figure 36

Prometheus primarily excels in metrics collection and storage, while Grafana specializes in advanced visualization and dashboarding. Grafana enhances Prometheus by providing rich graphical representations essential for effective monitoring, analysis, and decision-making.

Grafana:

Grafana is a leading open-source platform for monitoring and observability, specializing in visualizing time series data. It integrates with various data sources, including databases, cloud services, and monitoring systems like Prometheus, Influx DB, Elasticsearch, and more. Grafana's key features include:

- 1. Visualization:** It offers a wide range of visualization options such as graphs, charts, histograms, heatmaps, and geo-maps. Users can customize visualizations with annotations, thresholds, and alerts.
- 2. Dashboarding:** Grafana allows users to create and manage dashboards that consolidate metrics and logs from multiple sources. Dashboards can be shared, exported, and version-controlled for collaborative use.

3. **Alerting:** Built-in alerting functionalities notify users via email, Slack, or other channels based on defined thresholds and conditions. Alerts can be configured directly within Grafana based on data from connected data sources.
4. **Plugins and Integrations:** Grafana's plugin ecosystem extends its capabilities with additional data sources, panel types, and authentication methods. This extensibility allows customization and integration into diverse monitoring environments.
5. **Community and Support:** Grafana benefits from an active community that contributes plugins, dashboards, and support resources. It is widely adopted across industries for monitoring and managing complex infrastructures and applications.

Overall, Grafana empowers users with intuitive, interactive visualizations and powerful dashboarding capabilities, making it a preferred choice for monitoring and observability needs in both small-scale deployments and large-scale enterprise environments.

Prometheus Integration with Grafana:

Prometheus integrates seamlessly with Grafana, serving as a primary data source. Grafana leverages Prometheus' time series data to create detailed graphs, charts, and dashboards. This integration enables robust visualization of metrics collected by Prometheus, enhancing monitoring and observability. Grafana's alerting features can also utilize Prometheus alert rules, notifying stakeholders of critical events based on predefined thresholds and conditions, thereby facilitating proactive incident response and system management.

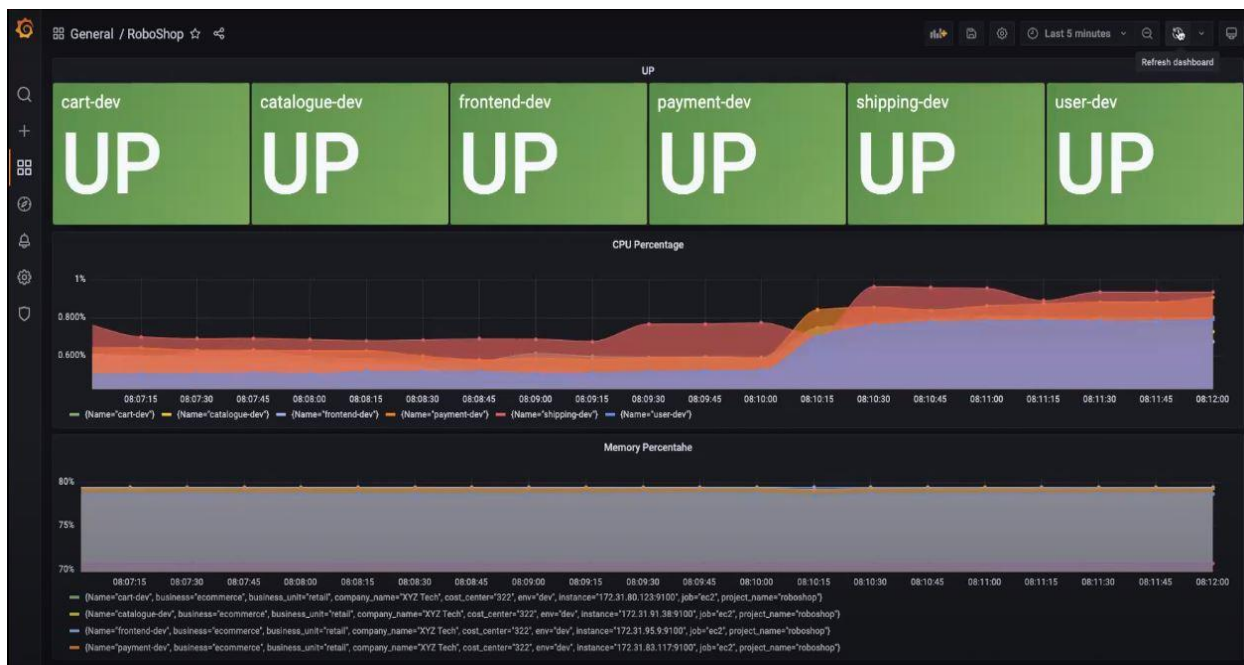


Figure 37

In this Grafana dashboard, we display the status (UP/DOWN) of Component API Servers in both dev and prod environments, along with CPU and Memory percentages. These metrics help fulfill the saturation signal from the 4 golden rules for monitoring

*Note: We have scaled up pod resources to meet demand, adjusting allocations as necessary. Developers synchronize databases with applications using caching mechanisms to optimize performance further. This proactive approach reduces downtime and latency, ensuring efficient resource utilization and maintaining optimal system performance.

Application Performance Monitoring (APM):

Application Performance Monitoring (APM) is a practice and set of tools used to ensure the optimal performance and availability of applications. APM solutions typically monitor and analyze key metrics such as response times, error rates, and resource usage within applications. They provide insights into application behavior, identify performance bottlenecks, and help troubleshoot issues affecting user experience. APM tools often include features for real-time monitoring, transaction tracing, code-level visibility, and correlation of performance data with business metrics. By proactively monitoring applications, APM helps organizations maintain high availability, improve responsiveness, and deliver a seamless user experience while supporting effective DevOps and application lifecycle management practices.

New Relic:

New Relic is a prominent Application Performance Monitoring (APM) and observability platform designed to help organizations monitor, troubleshoot, and optimize the performance of their applications and infrastructure in real-time.

Key Features:

1. **Monitoring:** New Relic provides comprehensive monitoring capabilities across applications, microservices, containers, and infrastructure. It tracks key performance indicators such as response times, throughput, error rates, and resource utilization.
2. **Visualization and Dashboards:** It offers customizable dashboards and visualizations that allow users to monitor and analyze data from various perspectives, helping teams gain insights into application performance trends and anomalies.
3. **Alerting and Notification:** New Relic enables proactive monitoring through customizable alerts and notifications based on predefined thresholds and conditions. This ensures that teams can respond promptly to performance issues before they impact users.

4. **Application Mapping:** It automatically maps application dependencies and provides visibility into complex architectures, facilitating efficient troubleshooting and root cause analysis.
5. **Integration:** New Relic integrates with a wide range of technologies including cloud platforms (AWS, Azure), databases (MySQL, PostgreSQL), and third-party services, allowing for seamless data aggregation and correlation across the stack.
6. **Analytics:** The platform includes advanced analytics capabilities to correlate performance data with business metrics, enabling organizations to align technical performance with business outcomes.
7. **Synthetic Monitoring:** New Relic offers synthetic monitoring to simulate user interactions and measure application performance from various global locations.

Overall, New Relic helps organizations improve application reliability, optimize performance, and deliver exceptional digital experiences by providing comprehensive visibility and actionable insights into their IT environments.

New Relic and Kubernetes Cluster (EKS) integration:

New Relic provides robust integration capabilities with Kubernetes, enhancing monitoring and observability in containerized environments:

1. **Kubernetes Cluster Monitoring:** New Relic can monitor the overall health and performance of Kubernetes clusters, including nodes, pods, and deployments. It provides insights into resource utilization, workload distribution, and cluster capacity.
2. **Pod-Level Monitoring:** New Relic captures detailed metrics from individual pods running within Kubernetes clusters. This includes metrics such as CPU usage, memory consumption, network traffic, and container health.
3. **Dynamic Service Discovery:** New Relic automatically discovers and monitors Kubernetes services and applications, adapting to changes in deployments and scaling events without manual configuration.
4. **Alerting and Notification:** It offers configurable alerts based on Kubernetes metrics and events, notifying teams of performance degradation, resource constraints, or other issues affecting Kubernetes workloads.
5. **Visualization and Dashboards:** New Relic provides customizable dashboards and visualizations to monitor Kubernetes metrics and application performance, enabling teams to analyze trends, identify anomalies, and troubleshoot efficiently.
6. **Integration with Prometheus:** New Relic integrates with Prometheus, allowing users to leverage Prometheus metrics within the New Relic platform for unified monitoring and correlation with other data sources.

Overall, New Relic's integration with Kubernetes enhances visibility, scalability, and reliability in containerized environments, supporting organizations in optimizing application performance and ensuring seamless operation of Kubernetes deployments.

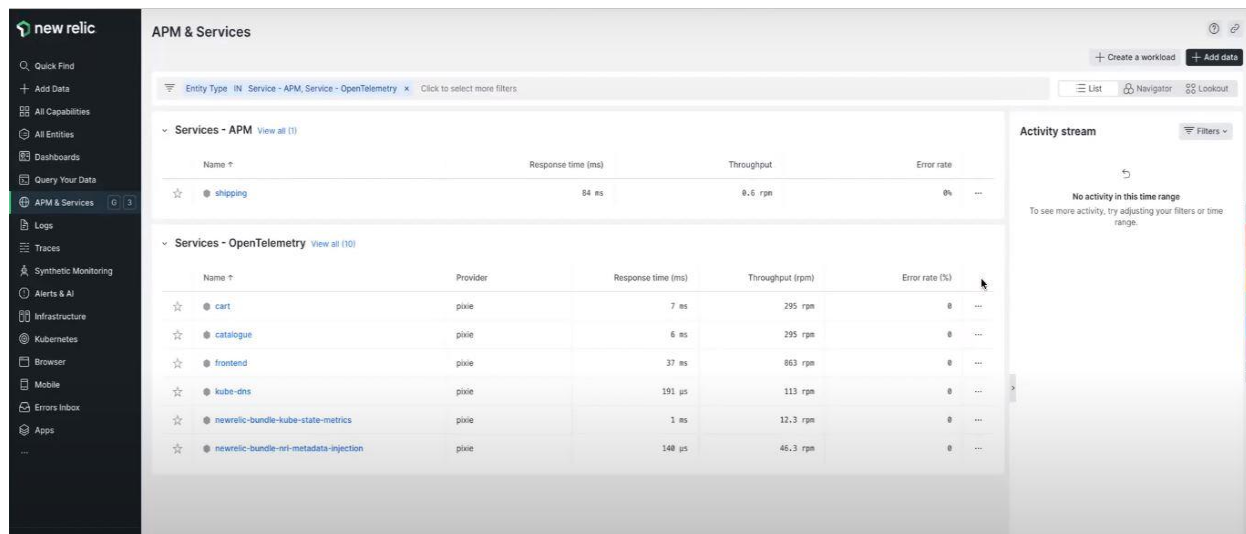


Figure 38

*Note: In our DevOps project, the Shipping API service is experiencing extended load times and integration challenges with other components. To investigate further, we'll utilize an APM tool to analyze the root cause of latency issues specifically within the Shipping API. While general integration provides default logs for all microservices, the Shipping service, which employs both JAVA and Node.js, requires additional support from New Relic to effectively capture and analyze these metrics.

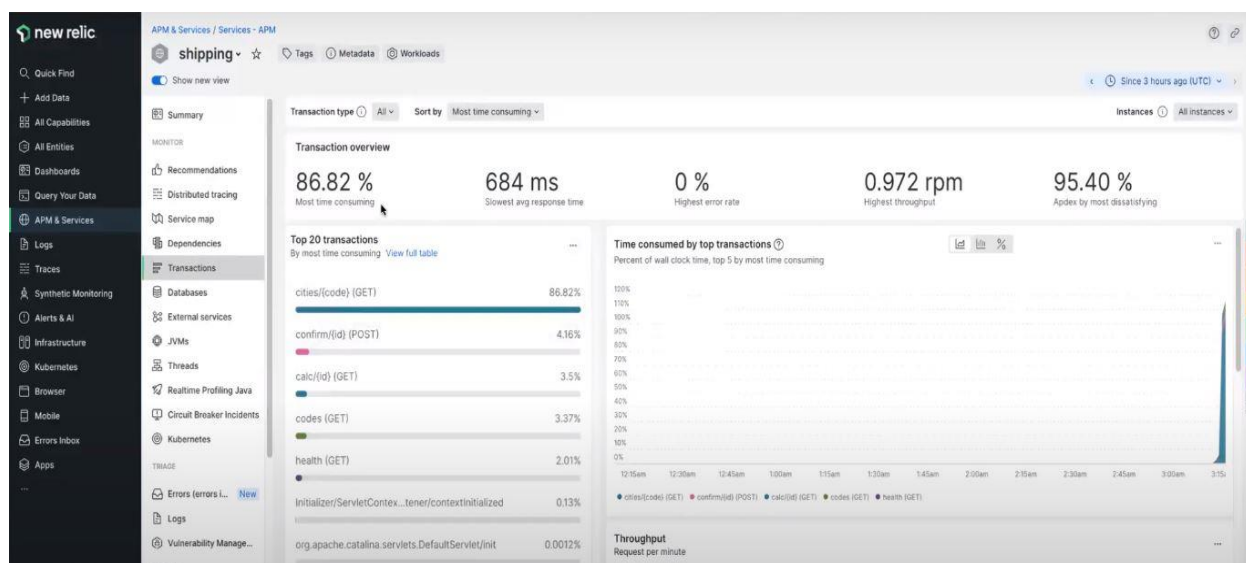


Figure 39

*Note: Further investigation revealed that the cities schema in MySQL is causing delays, impacting overall performance. Addressing the inefficiencies in loading this schema is crucial to improving the speed and responsiveness of the Shipping API service in our DevOps project.

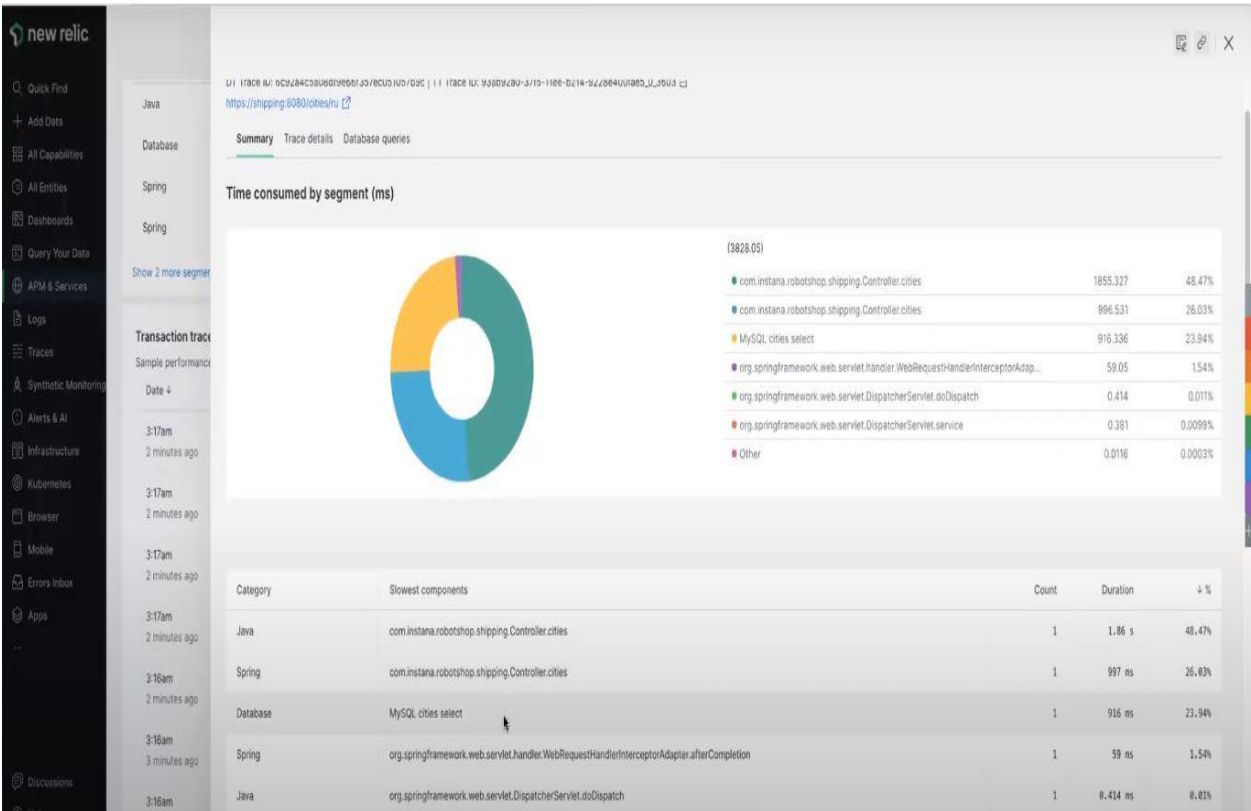


Figure 40

Development and Deployment Strategies:

There are of 3 types of strategies, they are

1. **Mutable:** Used for dynamic updates, allowing for real-time changes and adjustments to applications without the need for complete redeployment.
2. **Immutable:** Ensures consistent deployment by using fixed versions of applications, reducing variability, and enhancing reliability across environments.
3. **Containers:** Facilitates consistent deployment by encapsulating applications and their dependencies, ensuring they run uniformly across different environments.

These strategies optimize the process of application building, deployment, and maintenance, balancing the need for flexibility with the requirement for predictability. This helps meet diverse operational and development needs in various environments, ensuring efficient and reliable application performance.

Mutable Infrastructure

Definition: Mutable infrastructure allows changes and updates to be applied directly to the existing environment. This means configurations, applications, and infrastructure components can be modified after their initial deployment.

Use Cases:

- **Development Environments:** Frequent updates and configuration changes are necessary.
- **Quick Fixes:** Immediate patches or hotfixes that need to be applied without redeploying the entire infrastructure.

Advantages:

- **Flexibility:** Easier to make quick changes without needing to redeploy everything.
- **Less Downtime:** Immediate changes can reduce the downtime compared to redeploying.

Disadvantages:

- **Configuration Drift:** Over time, the environment might diverge from the original configuration, leading to inconsistencies.
- **Complex Debugging:** Harder to track changes and diagnose issues due to continuous modifications.

Best Practices:

- **Configuration Management Tools:** Use tools like Ansible, Puppet, or Chef to manage changes systematically.
- **Version Control:** Keep track of configuration changes using version control systems.

Immutable Infrastructure

Definition: Immutable infrastructure involves deploying new versions of servers or components instead of updating existing ones. Once deployed, the infrastructure is not changed; instead, it is replaced with a new instance containing the updated configuration or code.

Use Cases:

- **Production Environments:** Ensures stability and consistency by avoiding direct changes.
- **Disaster Recovery:** Easy to revert to a previous stable version if something goes wrong.

Advantages:

- **Consistency:** Each deployment starts from the same baseline, ensuring a known state.
- **Simpler Rollbacks:** Reverting to a previous state is straightforward by redeploying the last known good configuration.
- **Reduced Configuration Drift:** Eliminates the risk of unexpected changes over time.

Disadvantages:

- **Resource Intensive:** Can be resource-intensive as entire instances are replaced rather than updated.
- **Deployment Complexity:** Requires robust deployment pipelines to handle frequent redeployments.

Best Practices:

- **Use Infrastructure as Code (IaC):** Tools like Terraform, CloudFormation ensure environments are defined and deployed consistently.
- **Automated Deployment Pipelines:** Implement CI/CD pipelines to automate the creation and deployment of new instances.
- **Blue-Green Deployment:** Implement strategies like blue-green or canary deployments to minimize downtime and risks during updates.

Containers

Definition: Containers encapsulate an application and its dependencies in a lightweight, standalone package that can run consistently across different computing environments.

Use Cases:

- **Microservices Architecture:** Each service can be developed, deployed, and scaled independently.
- **Development and Testing:** Ensures consistency across development, staging, and production environments.
- **Scalable Deployments:** Facilitates scaling applications horizontally.

Advantages:

- **Consistency:** Containers ensure that the application runs the same way regardless of the environment.

- **Isolation:** Each container runs in its own isolated environment, reducing conflicts and improving security.
- **Portability:** Containers can run on any system that supports the container runtime, such as Docker.
- **Resource Efficiency:** Containers share the host system's kernel, making them more lightweight compared to virtual machines.

Disadvantages:

- **Complexity:** Managing large numbers of containers can be complex without proper orchestration.
- **Security:** Containerized environments need robust security practices to ensure isolation and prevent breaches.

Best Practices:

- **Use Docker for Containerization:** Define container images using Dockerfiles to ensure consistency.
- **Container Orchestration:** Use Kubernetes or Docker Swarm to manage container deployment, scaling, and operations.
- **Monitoring and Logging:** Implement monitoring (e.g., Prometheus, Grafana) and logging (e.g., ELK Stack) for containerized applications.
- **CI/CD Integration:** Integrate containers into CI/CD pipelines to automate the build, test, and deployment processes.

Integration of Mutable, Immutable Infrastructure, and Containers in DevOps

Dynamic Updates (Mutable):

- Use in environments where flexibility and quick updates are required, like development and testing environments.
- Employ configuration management tools (e.g., Ansible, Puppet) to manage changes systematically and track them using version control.

Stability (Immutable):

- Use in production environments to ensure stability and consistency.

- Implement deployment strategies like blue-green and canary deployments to minimize risks and downtime.
- Use Infrastructure as Code (IaC) to define and manage infrastructure, ensuring every deployment is consistent and traceable.

Consistent Deployment (Containers):

- Use containers to ensure applications run consistently across different environments.
- Employ container orchestration tools (e.g., Kubernetes) to manage deployment, scaling, and maintenance of containerized applications.
- Integrate containers with CI/CD pipelines for automated, reliable, and consistent deployments.

Combined Strategy Example:

1. Development Phase:

- Use mutable infrastructure for dynamic updates.
- Developers use Docker containers to ensure their applications run consistently in any environment.

2. Testing Phase:

- Transition to immutable infrastructure for staging environments.
- Deploy applications using container orchestration to simulate production environments.

3. Deployment Phase:

- Use immutable infrastructure for production environments to ensure consistency and stability.
- Deploy applications as containers using Kubernetes, employing strategies like blue-green deployments for zero-downtime updates.

4. Operations Phase:

- Continuously monitor and log container performance.
- Use IaC tools to manage infrastructure and ensure consistent deployments.

By integrating mutable infrastructure for flexibility, immutable infrastructure for stability, and containers for consistent deployments, DevOps teams can achieve a balanced, efficient, and reliable software delivery process.

*Note: In our DevOps project, we are developing and deploying our application using a container approach to ensure consistent deployment. This strategy encapsulates the application and its dependencies, providing uniform performance across different environments.

Deployment Strategy	Containers
What is this?	<p>Containers are a modern approach to running applications in isolated environments. By default, containers are immutable, meaning new containers are created whenever there is a change.</p> <p>Approach: To manage containers, we are using Kubernetes orchestrator via AWS EKS.</p>
Application Load	<p>50 concurrent users.</p> <p>80 TPS (Transactions Per Second)</p>
Deployment Strategy	Kubernetes Deployment by default uses Rolling Updates.
Errors	Zero (0)
Deployment Time	All Pods =~ 1min
Scaling	HPA (Horizontal Pod Autoscaling): The scaling is much faster because these are containers.
High Availability	We are using servers from two Availability Zones (AZ) because we have two subnets in different AZs.
Roll Back	Rolling back to an old version is done by redeploying the previous version. The operation cost is low since it requires minimal effort.

Accessing the Application: Screenshots of User Accessibility Post Deployment:

Screenshot displaying all the servers that are used in our Roboshop Project:

The screenshot shows the AWS Management Console for the us-east-1 region. The main content area displays the 'Instances (6)' page. The left sidebar contains the navigation menu with 'Instances' selected. The top bar shows the AWS logo, search bar, and user profile.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability
elk	i-03abcc9bc70f681ee	Stopped	t3.large	-	View alarms +	us-east-1a
prometheus	i-036af3b10f65f1d6b	Stopped	t3.small	-	View alarms +	us-east-1c
jenkins	i-024574c705ec281f0	Stopped	t3.small	-	View alarms +	us-east-1c
sonarqube	i-0493320500e33c50d	Stopped	t3.medium	-	View alarms +	us-east-1c
workstation	i-0298747a8f2e28695	Stopped	t3.small	-	View alarms +	us-east-1c
nexus	i-0317a4cad7b8c8ac4	Stopped	t3.medium	-	View alarms +	us-east-1b

Below the table, there is a 'Select an instance' dialog box.

Figure 41

Screenshot displaying the Jenkins CI/CD pipeline for different API services in Roboshop Project

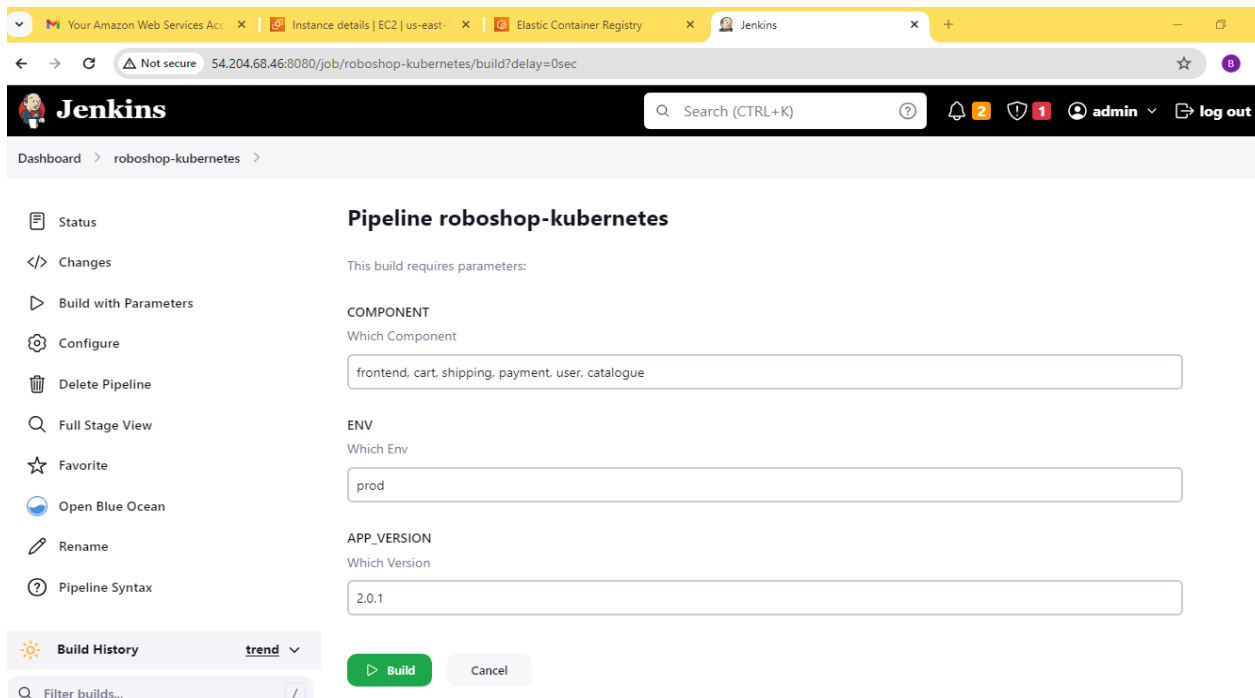


Figure 42

Screenshot displaying the running pods in the K9'S Tool

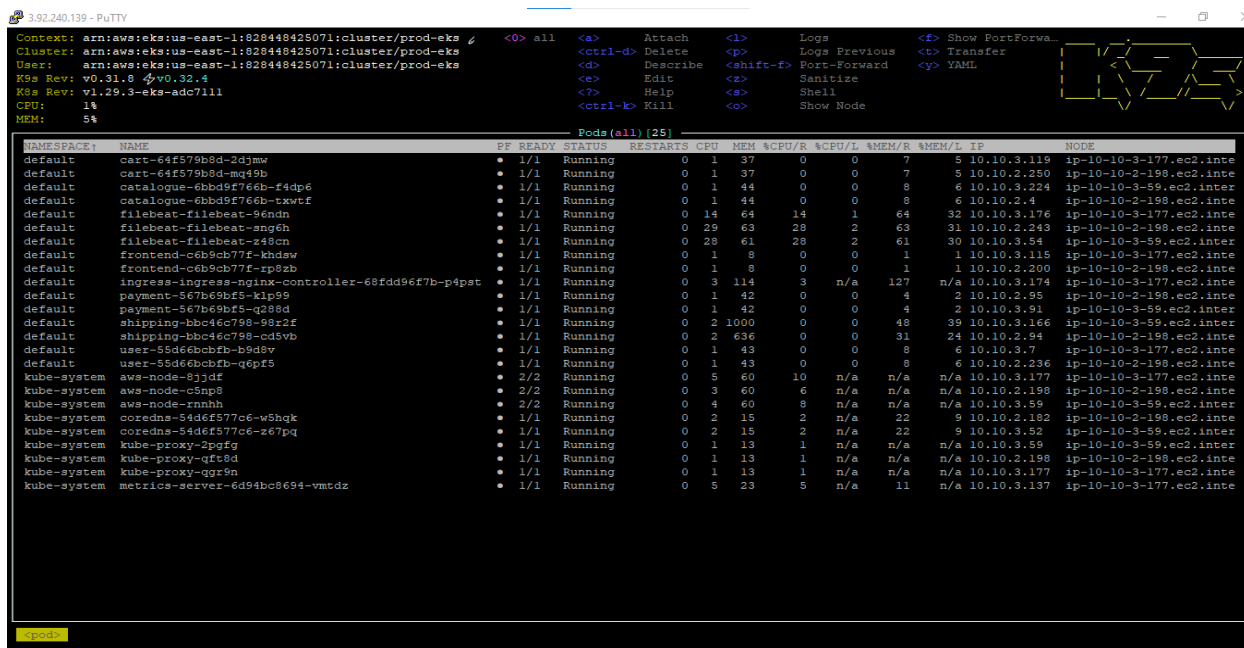
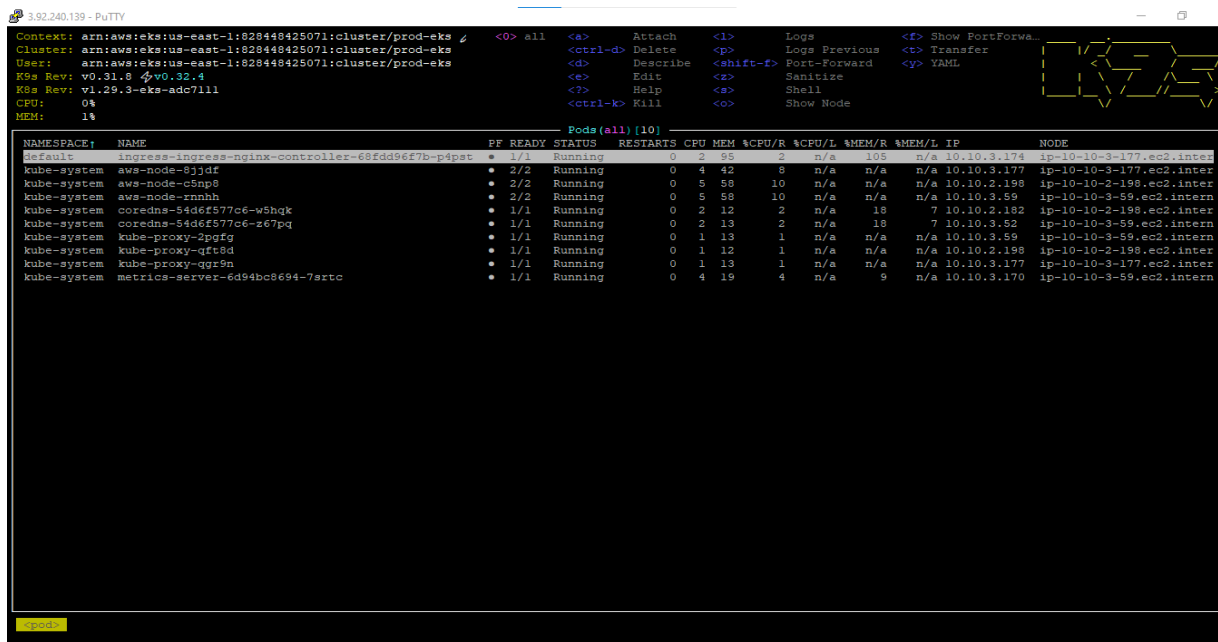


Figure 43

Screenshot displaying the running pods in the K9's tool.



The screenshot shows the K9's tool interface, a terminal-based Kubernetes cluster explorer. At the top, it displays the context (arn:aws:eks:us-east-1:828448425071:cluster/prod-eks), cluster name, and user. Below this, a table lists the running pods. The table has columns for Namespace, Name, Phase, Ready, Status, Restarts, CPU, Memory, and Node. The pods listed include ingress-ingress-nginx-controller, kube-system pods (aws-node, coredns, kube-proxy, metrics-server), and a metrics-server pod.

NAMESPACE	NAME	PHASE	READY	STATUS	RESTARTS	CPU	MEM	NODE
default	ingress-ingress-nginx-controller-68fdd96f7b-p4p4t	Running	1/1	Running	0	2	95	ip-10-10-3-177.ec2.intern
kube-system	aws-node-8jgdf	Running	2/2	Running	0	4	42	ip-10-10-3-177.ec2.intern
kube-system	aws-node-c8np8	Running	2/2	Running	0	5	58	ip-10-10-2-198.ec2.intern
kube-system	aws-node-zmnhh	Running	2/2	Running	0	5	58	ip-10-10-3-59.ec2.intern
kube-system	coredns-54d6f577c6-w5hgk	Running	1/1	Running	0	2	12	ip-10-10-2-182.ec2.intern
kube-system	coredns-54d6f577c6-z67pq	Running	1/1	Running	0	2	13	ip-10-10-3-59.ec2.intern
kube-system	kube-proxy-2pgfg	Running	1/1	Running	0	1	13	ip-10-10-3-59.ec2.intern
kube-system	kube-proxy-qft8d	Running	1/1	Running	0	1	12	ip-10-10-2-198.ec2.intern
kube-system	kube-proxy-qgr9n	Running	1/1	Running	0	1	13	ip-10-10-3-177.ec2.intern
kube-system	metrics-server-6d94bc8694-7sttc	Running	1/1	Running	0	4	19	ip-10-10-3-59.ec2.intern

Figure 44

Screenshot displaying Roboshop landing page and list of Categories.

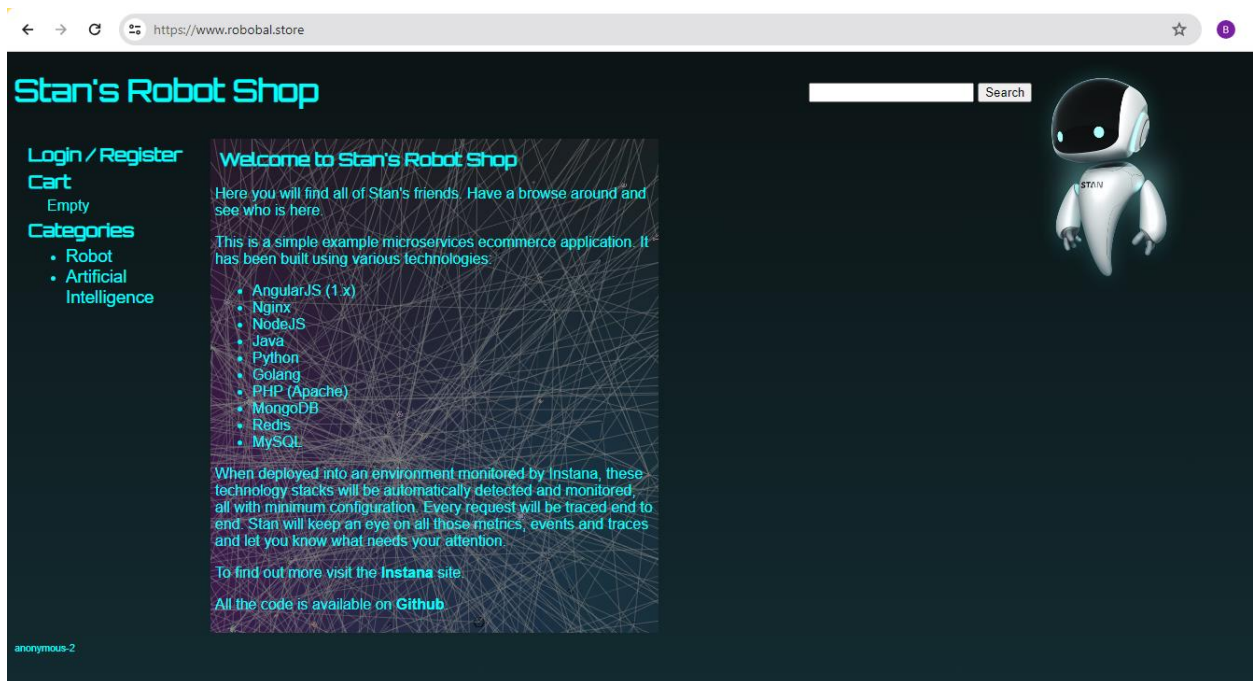


Figure 45

Screenshot displaying the Login/Register page to register the user to use Roboshop Application.

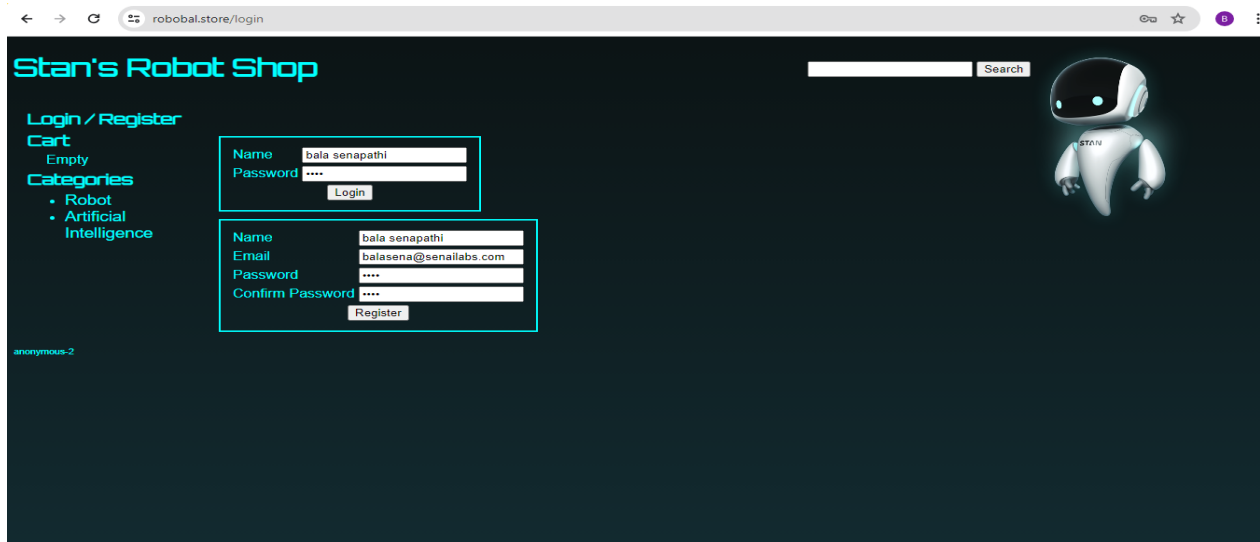


Figure 46

Screenshot displaying the registered user information after login into the Roboshop Application.

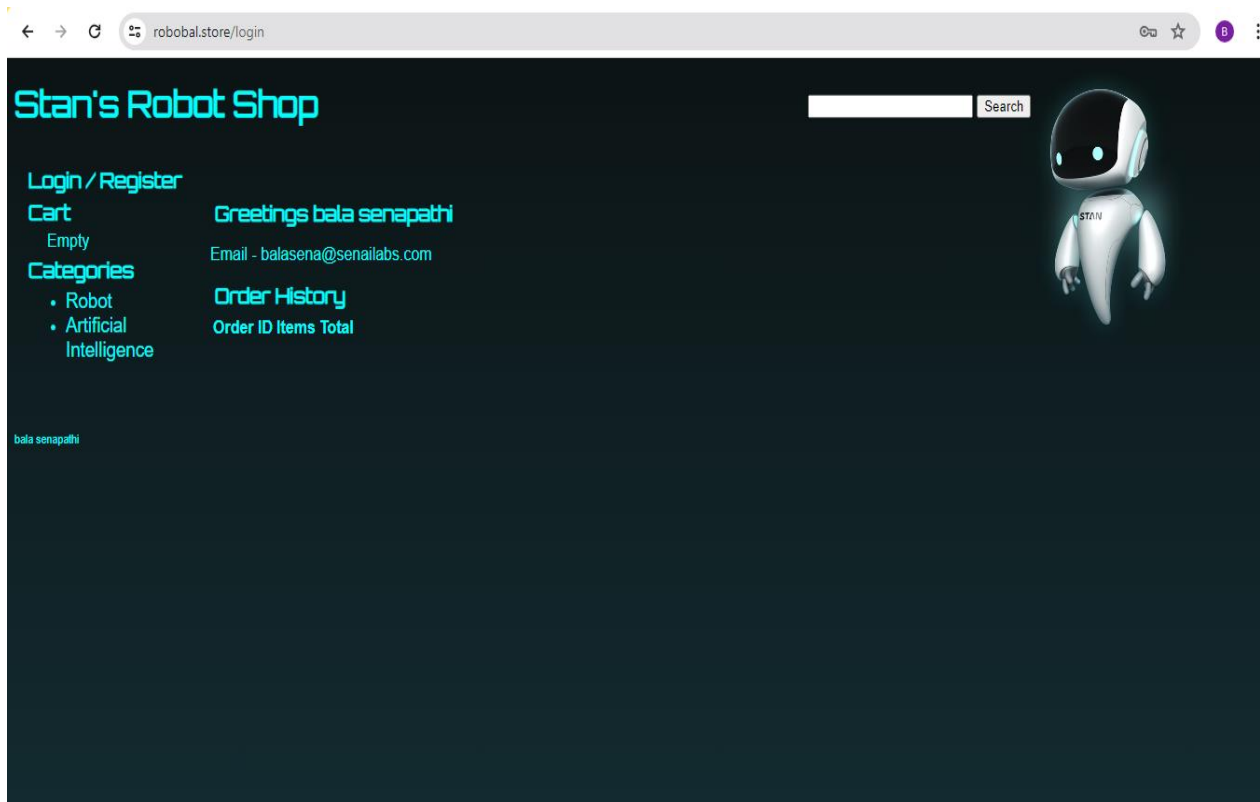


Figure 47

Screenshot displaying the empty cart for user Bala Senapathi before making the order.

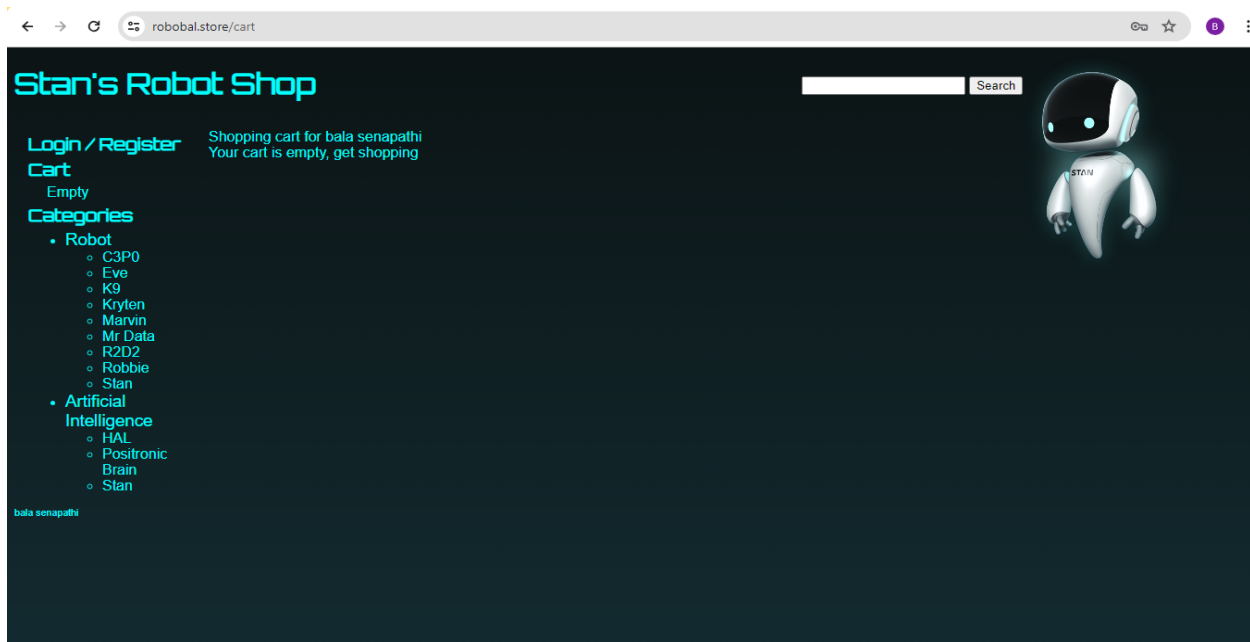


Figure 48

Screenshot displaying the addition of items to the user Bala Senapathi cart from list of categories.

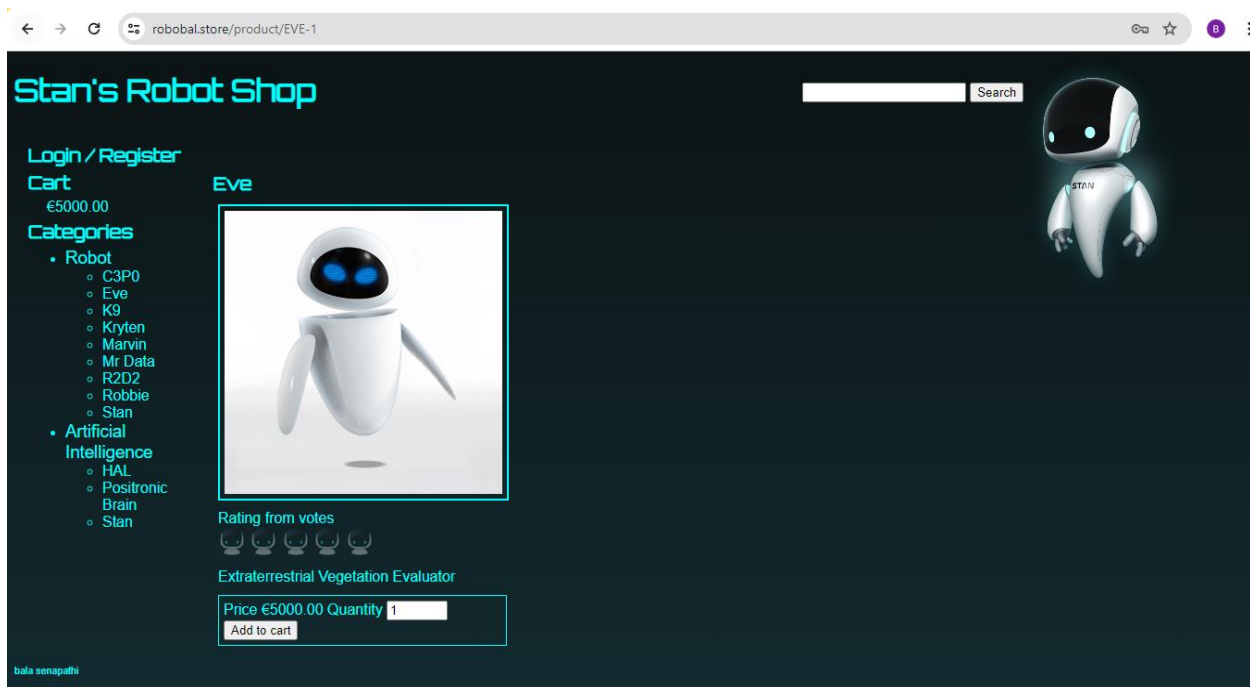


Figure 49

Screenshot displaying the Shopping cart for the user bala senapathi.



Figure 50

Screenshot displaying the shipping information across the world with country and location info.

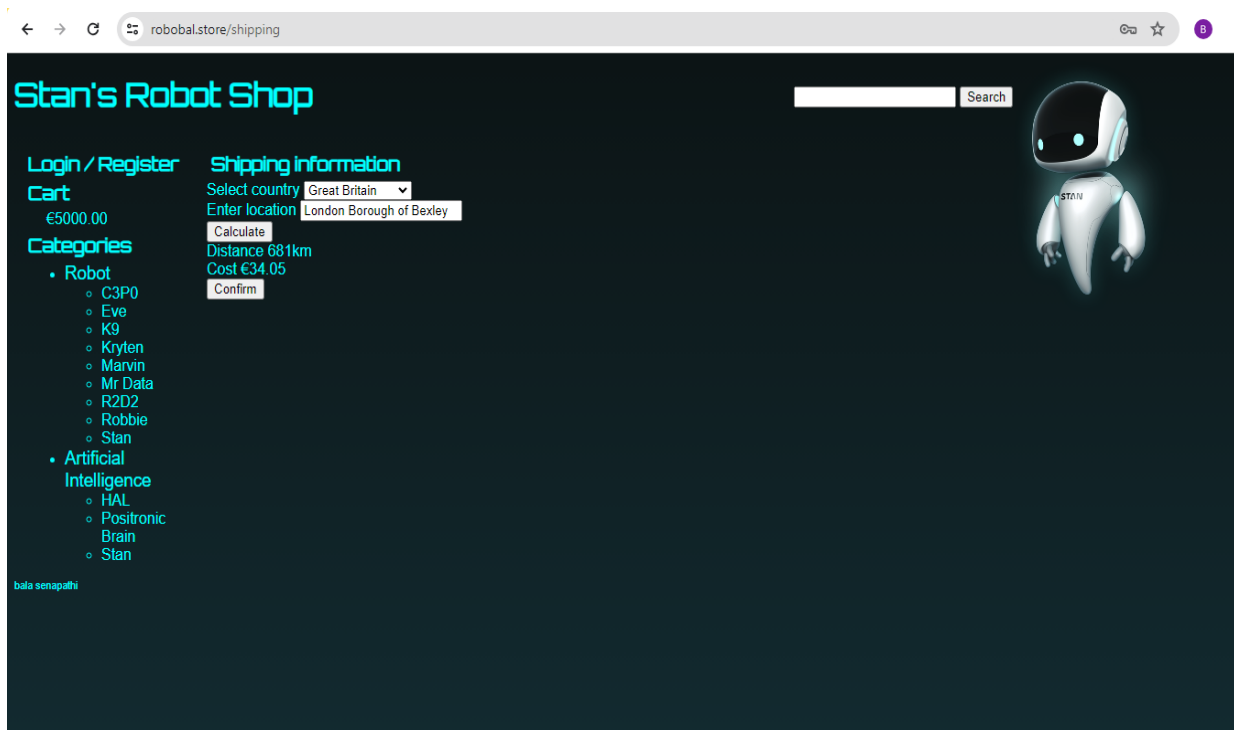


Figure 51

Screenshot displaying the Review of order with Quantity, Item Name and Total Price.

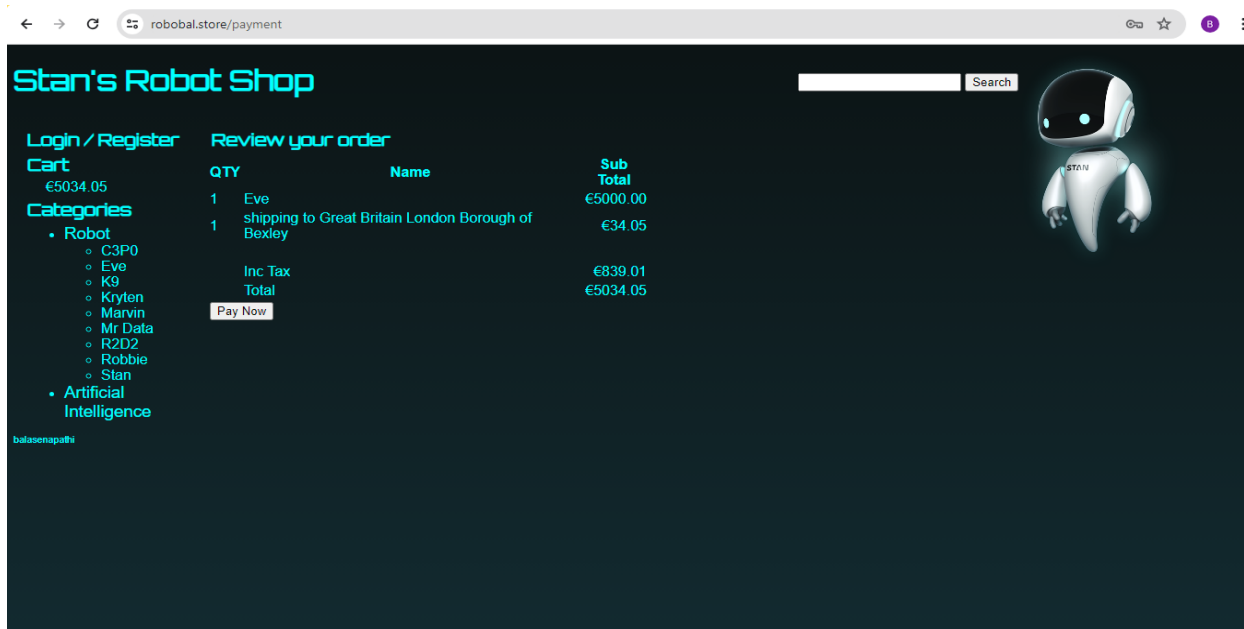


Figure 52

Screenshot displaying the payment confirmation with placed order and Order ID details.

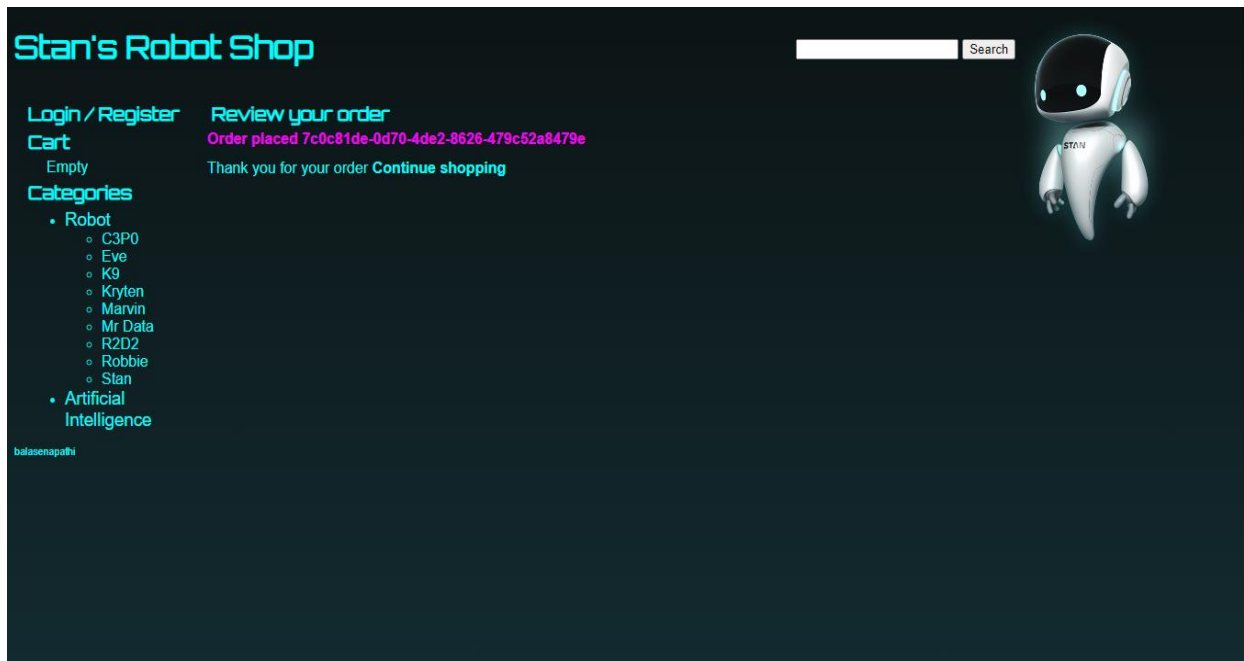


Figure 53

Screenshot displaying the order history with Order ID, Items, Total price with shipping details.

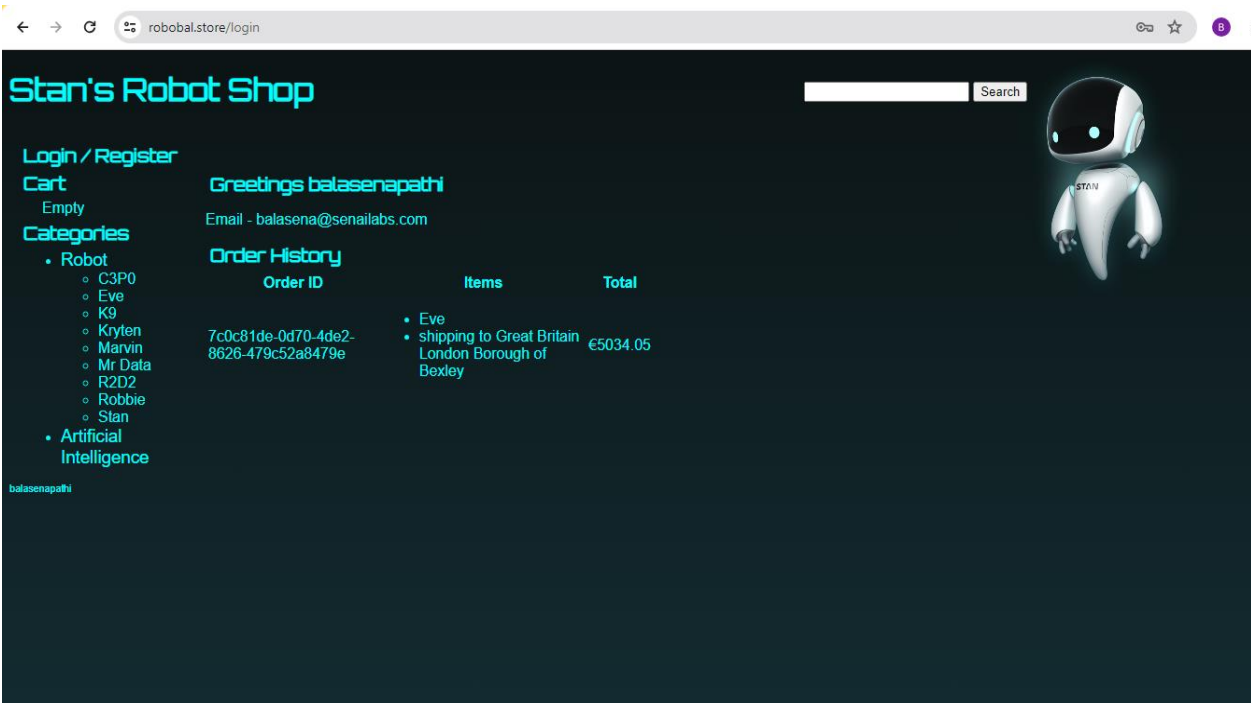


Figure 54

Screenshot displaying the Latency, Traffic, Errors for different API's using container approach.

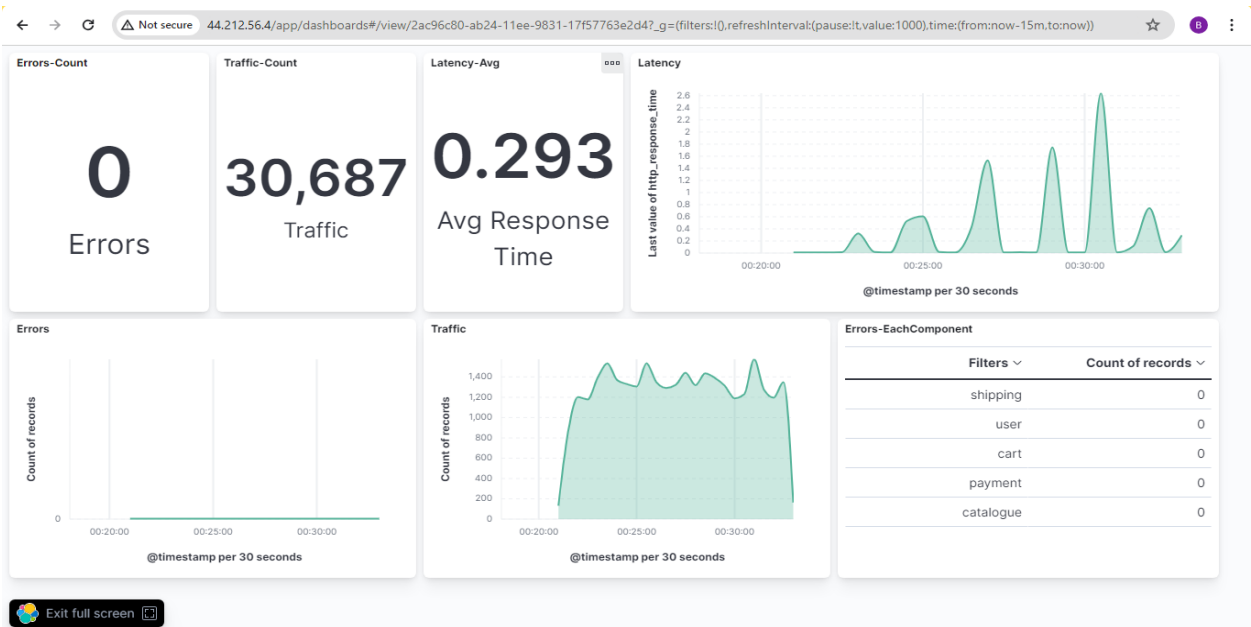


Figure 55

Screenshot displaying the Saturation for Server API's, CPU and Memory Percentage usage.



Figure 56

Conclusion

In our Roboshop project, we used a container-based approach with Kubernetes via AWS EKS for consistent deployments. Key tools included Terraform for IaC, Ansible for configuration management, Jenkins for CI/CD, SonarQube for code quality analysis, Nexus for artifact management, ECR for container registry, and Helm for managing Kubernetes applications. We also used Prometheus and Grafana for monitoring, the ELK Stack for logging, and New Relic for APM. Horizontal Pod Autoscaling (HPA) provided faster scaling, and using servers from two Availability Zones (AZ) ensured redundancy. The rollback process was streamlined, reducing operational costs. These practices and tools enabled us to deliver a robust, scalable, and efficient e-commerce platform, ensuring a seamless experience for users.