DEVOPS AND DOCKER CONTAINERS

SOLUTION BY: TEAM SAI-INFRA

Aroop Kumar Gochhayat (DT- 20195040639)

Aparna Devi (DT-20195796307)

YOUTUBE	https://youtu.be/fCX6eTJuZpo
GITHUB	https://github.com/aroopkumargochhayat/inframind.git https://github.com/aroopkumargochhayat/shopizer.git
DOCKER	aroopkumargochhayat/mysql:latest aroopkumargochhayat/shopizer:latest

Introduction

Enterprises today face the challenge of rapidly changing competitive landscapes, evolving security requirements, and performance scalability, there's a huge gap between operations stability and rapid feature development.

Problems faced in traditional software developments:-

1. Applications developed using a single technology/platform is dependent on it for all of its functionalities. If there's a cross platform requirement then the platform should have the appropriate SDK's to support cross platform integration.
2. Application release is a very exhaustive process, it requires a lot of workforce and time investment.
3. Clashes between developers and release management team are imminent – "Breaks in Production...

   Dev: but it works fine in my machine."
4. Provisioning and maintenance of the IT infrastructure required by the application depending on its usage is a major challenge.

Devops saves the day:-

DevOps aims at coupling a tighter alignment between IT operations and businesses.

To address the issues above we make use of Devops.

- Application containerization is achieved with the help of **Docker Containers** we divide the application into web tier, app tier, database tier. By doing this we have the freedom of developing features of the application in any platform we desire and integrate it without affecting the stability of the application or its dependencies.

- Application release is taken care of by **Continuous integration and continuous delivery (CI/CD).** It is a practice that enables rapid software changes while maintaining system stability, scaling and security.
- IT infrastructure provisioning and maintenance is done with the help of **Kubernetes**. It takes care of the scaling and deployment of application in an IT environment.

THE BIG PICTURE

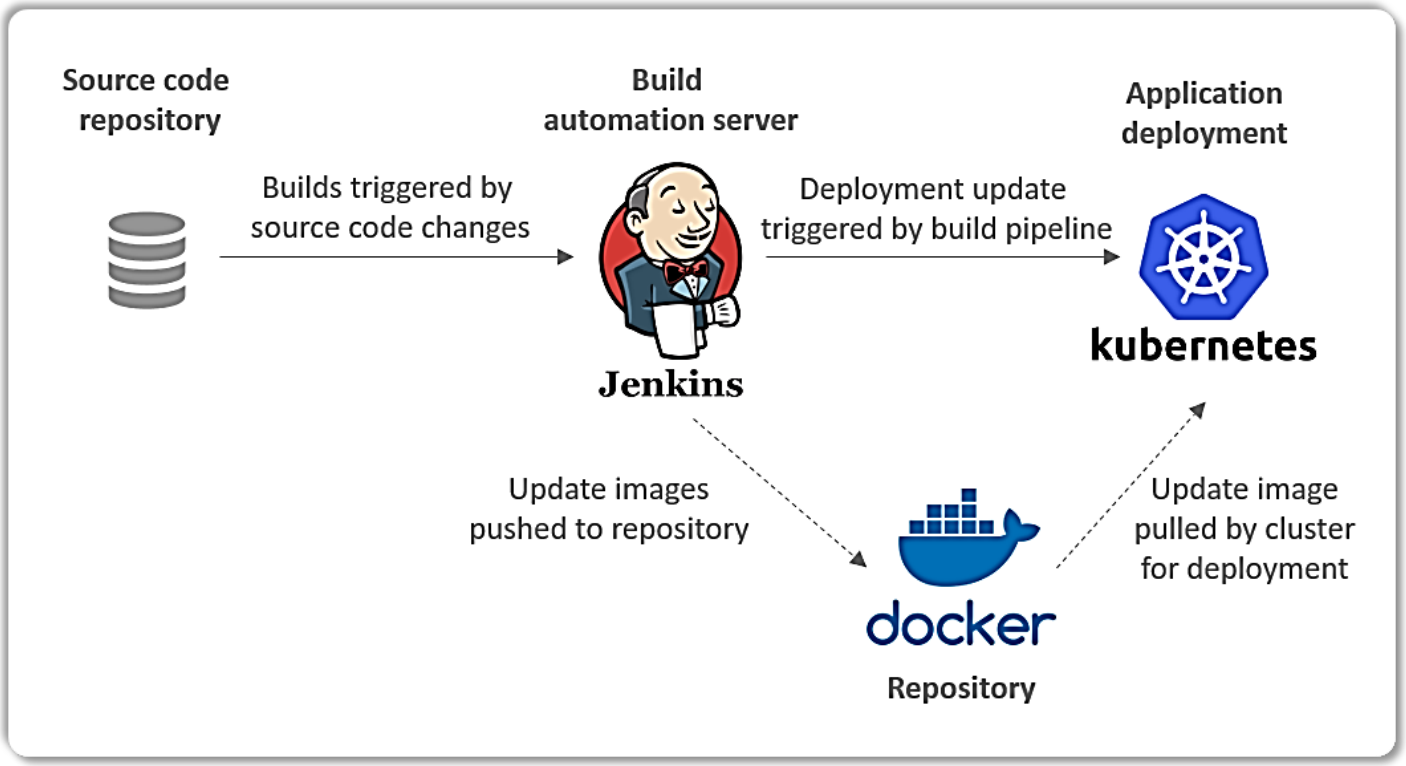
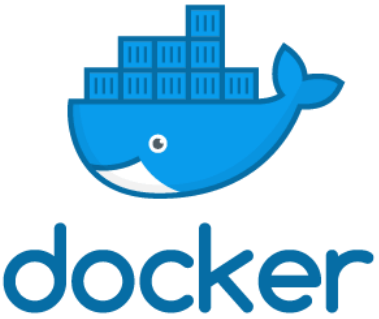


Fig. 1 Architectural overview

Technology Stack

1. Docker

Docker is a set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.



2. Git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

3. Maven

Maven is a project management and comprehension tool that provides developers a complete build lifecycle framework. Development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle.

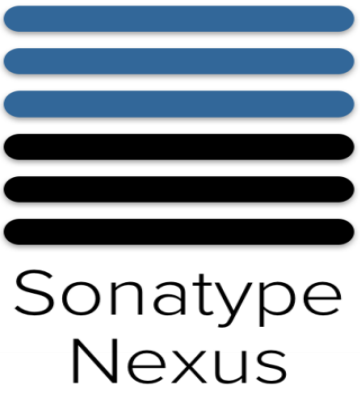
4. Jenkins

Jenkins is a free and open source automation server written in Java. Jenkins helps to automate the non-human part of the software development process, with continuous integration and facilitating technical aspects of continuous delivery.

5. Kubernetes

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.

6. Nexus



Nexus manages software "artifacts" required for development. If you develop software, your builds can download dependencies from Nexus and can publish artifacts to Nexus creating a new way to share artifacts within an organization.

7. Amazon Web Services

Amazon Web Services (AWS) is a subsidiary of Amazon that provides on-demand cloud computing platforms to individuals, companies, and governments, on a metered pay-as-you-go basis. In aggregate, these cloud computing web services provide a set of primitive abstract technical infrastructure and distributed computing building blocks and tools.

8. Kubernetes Operations (KOPS)

Kops is an official Kubernetes project for managing production-grade Kubernetes clusters. Kops is currently the best tool to deploy Kubernetes clusters to Amazon Web Services. The project describes itself as kubectl for clusters.

9. Grafana

Grafana is an open source metric analytics & visualization suite. It is most commonly used for visualizing time series data for infrastructure and application analytics but many use it in other domains including industrial sensors, home automation, weather, and process control.

Software/Hardware required

Software:

- Eclipse IDE for Java code development
- Git for SCM
- Web browser for inspecting changes

Hardware:

- Desktop/laptop with Linux/Windows OS
- Active internet connection

Achieved Cost Savings

On-Premise VS Cloud VS Virtualization:-

Note:- This cost estimation is based on restricted parameters and can be used as a reference to scale accordingly provided the parameters are met.

Source:- AWS TCO (Total Cost of Ownership), AWS Pricing.

PARAMETERS:-

1. Incoming traffic for the webservers are limited to 45-50 users for a small scale business.
2. Deployed web application front-end is developed on a JAVA platform with micro services like spring and hibernates.
3. MySQL database is used as the back-end for the application.
4. Minimal cost hardware components are used. Eg:- HDD is used in place of SSD
5. Virtualization is implemented on the cloud platforms hence costs are calculated with reference to the cloud providers
6. On-Premise servers are not considered virtualized so they don't contribute to performance enhancement and behave as typical servers hosted behind an ELB.
7. All the cloud cost estimates are done on an on-demand basis.
8. Free tier discounts are not considered in this estimate.
9. This is a **MONTHLY** cost estimate only.

RESOURCE	COUNT &SPECS	PRICING		
		ON-PREMISE	CLOUD	CLOUD VIRTUALIZATION
Load Balancer	1*(Classic ELB)	$2,237.194	$212.027	$18.38
Web-Server	2*(2 CPU,4G RAM)			$76.43
Database Server	1*(1 CPU, 4G RAM)			
Snapshot/Image	1*(AMI/DOCKER/PV)	$1,464.4	$4.76	$4.60
Volumes	2*(8G HDD) + 1*(10G HDD)			
Elastic IP/DNS	Min-1 Max-2	$11.00	$11.50	$3.60
IT Labor	DBAs, IT support	$73.138	$131.66	-
TOTAL		**$3,785.732**	**$359.947**	**$84.63**

- **You can save a whopping 97.7% of your costs on adapting cloud virtualization as compared to on-premise.**
- **On the other hand you'll save up to 76.48% of your costs as compared to VM's and Cloud IS.**

Architecture

The diagram below demonstrates the workflow of our solution.

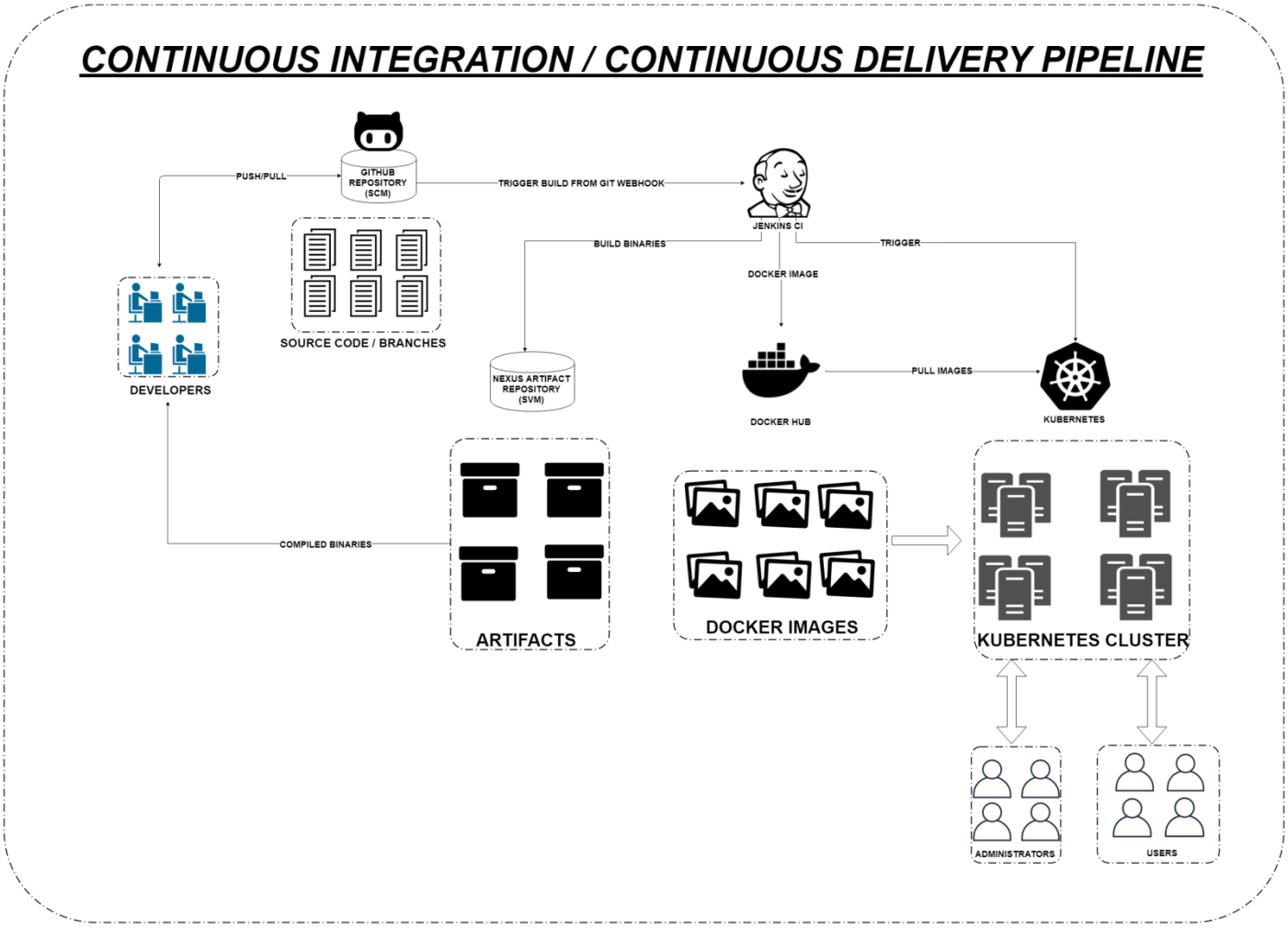


Fig. 2 Implemented architecture

Solution brief description

Objective of the solution is to implement a production grade environment to host a java web application and practice CI/CD to maintain the application with zero downtime.

The solution comprises of all the tools and a service listed above and uses them to implement a continuous integration and continuous delivery pipeline and a reliable infrastructure for production grade requirements.

P.S – The solution is divided into three phases, each phase is described below.

PHASE-1: - APPLICATION CONTAINERIZATION AND SOURCE CODE MANAGEMENT

SOURCE CODE BREIFING:-

We are using the open-source e-commerce application **SHOPIZER** as the required java code base. It is a java based web application with frameworks like spring and hibernate and uses **MAVEN** as a build tool for building the application.

The application is designed to use both H2 and MySQL database. However it comes prebuild with an H2 database and its required .db dump (SALESMANAGER.h2.db).

Issues with H2 database:-

- H2 is basically an in memory database for java based applications providing necessary JDBC connection APIs.
- H2 is a file system based database system, and it's not recommended in a production environment.

Resolution:-

- We need to migrate to a MySQL database.
- Configure necessary JDBC connection APIs in the source code to connect to our MySQL DB.
- Create necessary schema according to the requirements of the application.
- Deploy our DB engine in a persistent storage.

Bugs encountered:-

- During application initialization the hibernate framework always creates the required schema for the application to run.
- If the required schema is already present in the database then the initialization gets stuck at the schema creation step and the application never starts.
- You can check the official github repository of SHOPIZER for the path **sm-shop/src/main/resources/database.properties** at **Line No. 28** "hibernate.hbm2ddl.auto=update"

- When the above line is set to update it creates the schema but gets stuck on an existing schema.
- SOLUTION- set “hibernate.hbm2ddl.auto=none”. This will not create the schema every time the application is initialized
- You can take a backup of the created schema in the initial run of application, fix the bug and re-run the application with the schema uploaded to your desired migrated database.

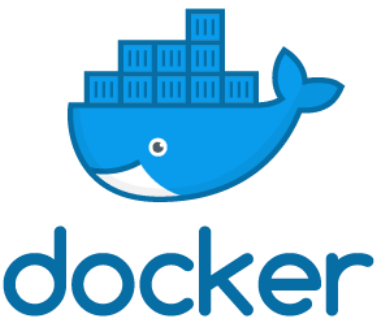
Source Code Management and build:-

- We are using github as our SCM repository, and Git integrated with eclipse IDE (e-GIT) for developer convenience of coding and pushing it to the repository.
- Docker Hub is our image repository for storing the custom images build by Dockerfile with proper image versioning using tags.
- Sonatype NEXUS is being used as our artifact repository for storing compiled binaries of the source code.
- MAVEN is being used to build the application.

Result:-

- The application is containerized into web tier, app tier and database tier.
- All of the source code is maintained in Github repos.
- We need to make custom docker images for both the application and the database consisting of the required dependencies, bug fixes and necessary schema.

TOOLS / SERVICES USED:-



Note: - Please refer to the github/docker hub repo links above for Phase-1 solution.

PHASE-2: - IMPLEMENTATION OF CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY PIPELINE FOR APPLICATION RELEASE PROCESS.

For implementing the CI/CD pipeline we completed the following objectives:-

1. Setting up Jenkins on a windows server
2. Automated trigger of pipeline upon code commit in VCS.
3. Automated building and testing of the source code.
4. Post successful build and test new docker build should be triggered and updated image pushed in the docker hub registry.
5. Parallel job should be done to update the NEXUS repo with new compiled binaries.
6. After docker image is pushed triggering a pull request from to kubernetes cluster to fetch the latest image and deploy it to the cluster.
7. All the Jobs should be parameterized to keep track of versioning of the deployed application.

SOLUTION:-

1. **Automated trigger of pipeline upon code commit in VCS** was achieved by using the git webhook plugin with the Jenkins CI server. One needs to select Git SCM in the build configuration and provide the git repo URL. Also you need to configure your git repo in repository settings to trigger the Jenkins webhook api on a code commit.
   Note: - You need to provide your GitHub credentials in Jenkins if you are using a private repository.

2. **Automated building and testing of source code** was done by maven as the java codebase has a pom.xml file which lists all the dependencies required to compile/build the project and test cases to test the project.
   If you are building a freestyle job then in the build section of the Jenkins job specify the command **mvn clean install**
   if you have **Maven Integration** plugin installed then just specify **clean install** in the Goals section.
   Note:-Maven binaries should be extracted and their path should be specified in the environment variables in the windows server.

3. **Docker building** was the most crucial and challenging task. As the devops engine was set up in a windows server to mimic a production environment and the server was provisioned on the cloud. We faced the following challenges:-
   - Windows docker installations cannot run linux Dockerfiles/Images so we can't build the dockerfile and push it to the repository from the Jenkins server.
   - Instances provisioned on the cloud don't give access to their BIOS settings to enable virtualization and we can't go for nested virtualization either.
   - The only instances which support Hyper-V (virtualization engine for windows) on AWS are i3.metal series. These are bare metal instances with virtualization enabled and costs $7.568/hr

   It was possible to build this solution in a physical environment with bare metal servers or with linux servers on cloud, but not with windows servers (except you are willing to pay $7.568/hr).

   WALKAROUND:-

   The only way to build the Dockerfiles was to do a cross platform build i.e. the codebase was build on the windows instance with maven and the workspace was copied to a linux instance with docker installed. We built and pushed the Dockerfile to the repository from the linux instance only.

   This was achieved by **Jenkins Master-Slave architecture** where the linux machine was added as a slave node to the windows Jenkins server through a JNLP connection and we used the archive workspace plugin to copy the entire windows workspace after build to the linux instance

   Note: - Jenkins need not to be installed in the slave machine to run the job. So this preserves the validity of the solution to use only windows server as the DevOps engine.

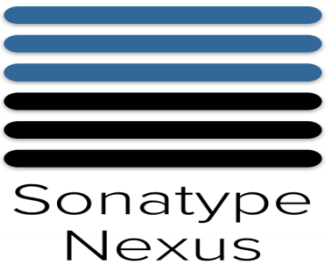
4. **Artifact Management** is done by pushing the build artifacts of maven into the artifact repositories with proper versioning tags. In this case we'll be using Sonatype NEXUS to manage our build artifacts. As soon as the build gets completed, both the docker file building and artifact pushing jobs are executed in parallels on different nodes to distribute the workload.

5. **Triggering kubernetes cluster to pull image from docker hub** was done by adding the kubernetes master node as a slave to the Jenkins server over JNLP and running kubectl commands in a pipeline.
   To be specific this command is used to pull the required image from the repo.

   Kubectl set image deployment/<deployment name> <deployment container name>=<docker repo name>/<application>:<tag id>

6. **Parameterized build** is done to maintain a consistent tag value which is necessary to know the version of image build and pushed to the repo, kubernetes further references to this version only to pull the image from docker hub repo. Thus the consistent tag value across the builds is passed through parameterized build and thereby maintaining image versioning.

TOOLS / SERVICES USED: -



PHASE-3: - DEPLOYMENT, MAINTENANCE AND MONITORING

Phase 3 objectives include:-

1. Provisioning a kubernetes cluster consisting of 1 master and 1 worker node.
2. Preserving state of the application.
3. Deployment of container images into kubernetes cluster environment
4. Load balancing, scaling, rollback, versioning, orchestration of the application.
5. Monitoring of cluster resources with a time-series database and visualization tool.

1. **Provisioning of the kubernetes cluster** was done on AWS cloud platform using the configuration management tool KOPS which deployed production grade kubernetes cluster. We have used one t2.micro instance with 64 GB GP2 SSD storage hosted as master and one t2.medium instance with 128 GB GP2 SSD storage. The server configurations, security groups, port opening, etc are specified in the github repositories.
2. **Stateful nature of the web application** is achieved by attaching a **persistent volume** to the clusters to use, we are not using simple volumes because they can't be dynamically provisioned. But with **persistent volume claim service** of kubernetes we can dynamically provision any type/size of volume we wish during runtime and attach it to the pods made by the deployments. So in case one of the pods fails the data will still be available and retained when pods are recreated.

3. **Deployment of images into kubernetes cluster** was done by making deployment configuration files (.yaml) which specified the container image to be used, number of replicas, services attached to expose those deployments, etc. The deployment architecture in the nodes can be visualized as follows:-

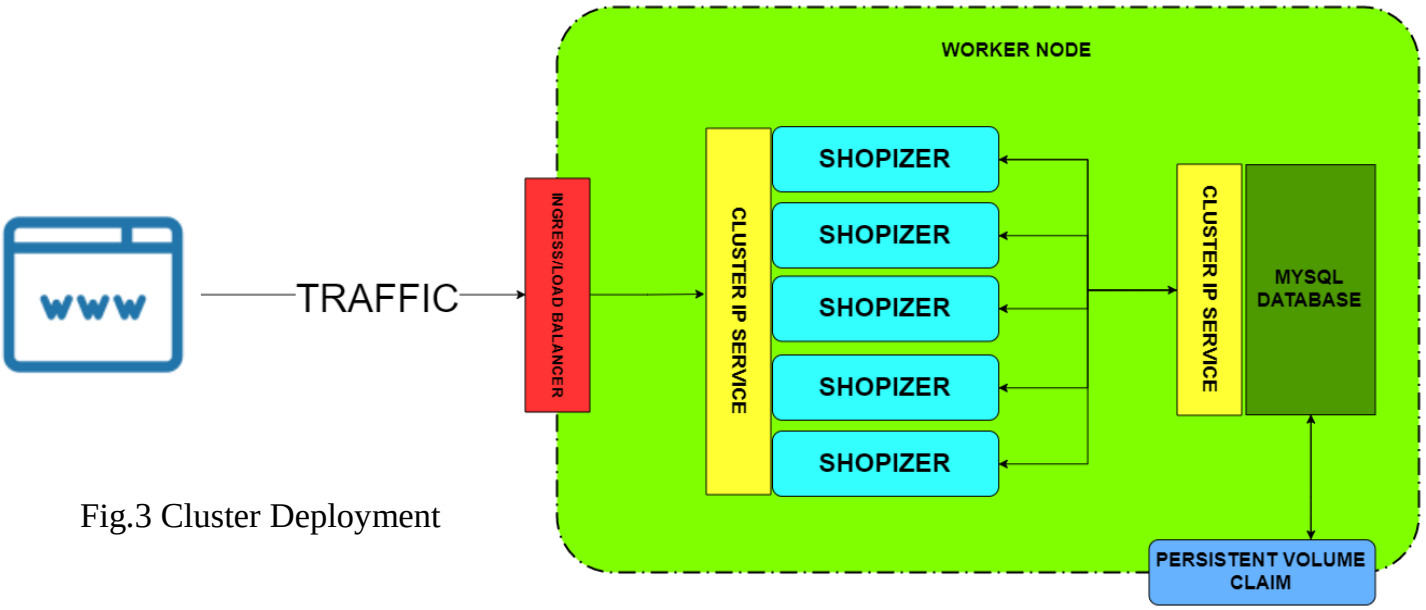


Fig.3 Cluster Deployment

4. **Maintenance of the clusters**, this is done by the kube-master which keeps track of the deployments, pods, nodes, service, volumes, replicas etc.

   The pods are provisioned by a deployment inside the cluster which makes a replica of the pods every time a deployment is created, and keeps track of the deployment history for rollback requirements.

   Load Balancing is done by the **Ingress-nginx** this is a project which provisions a classic load balancer and an nginx routing engine so that we can route requests to different api endpoints and ports depending on the structure of url and the load can be balanced amongst the replicas of application.

   Scaling is done by specifying the number of replicas to be maintained in the cluster deployment. The kube-master ensures that always the specified numbers of replicas are running in the cluster. This makes sure the application has zero downtime, i.e. even if a pod fails others are available to ingest the traffic to that pod until it is restarted by kube-master.

   Rollbacks of deployments are done by the replicas stored. Every deployment pod has a replica stored so in case a new deployment is done and it's unstable, we can always revert back to the previous deployment stored in the replicas.

Versioning of the application is being done in two aspects i.e. storing the binaries into the artifact repository and the image version in docker hub.

5. **Monitoring** is a crucial part of the kubernetes cluster and is being done in our project by using Amazon Cloudwatch and Grafana. The cloudwatch is a time-series database which stores the state of the resources provisioned on aws w.r.t time.

   Grafana is a visualization tool which uses cloudwatch as the data source and plots interactive graphs to represent the state of our clusters.

TOOLS / SERVICES USED: -

Scope of automation

The complete CI/CD pipeline provisioning can be automated by Ansible, an infrastructure configuration management tool featuring Infrastructure as a code (IaaC). The CI/CD infrastructure which takes hours, maybe days to build can be spinned up in minutes just by executing a simple ansible playbook which provisions all the required resources to make a CI/CD pipeline.

Here's a sample workflow of CI/CD using Ansible: -

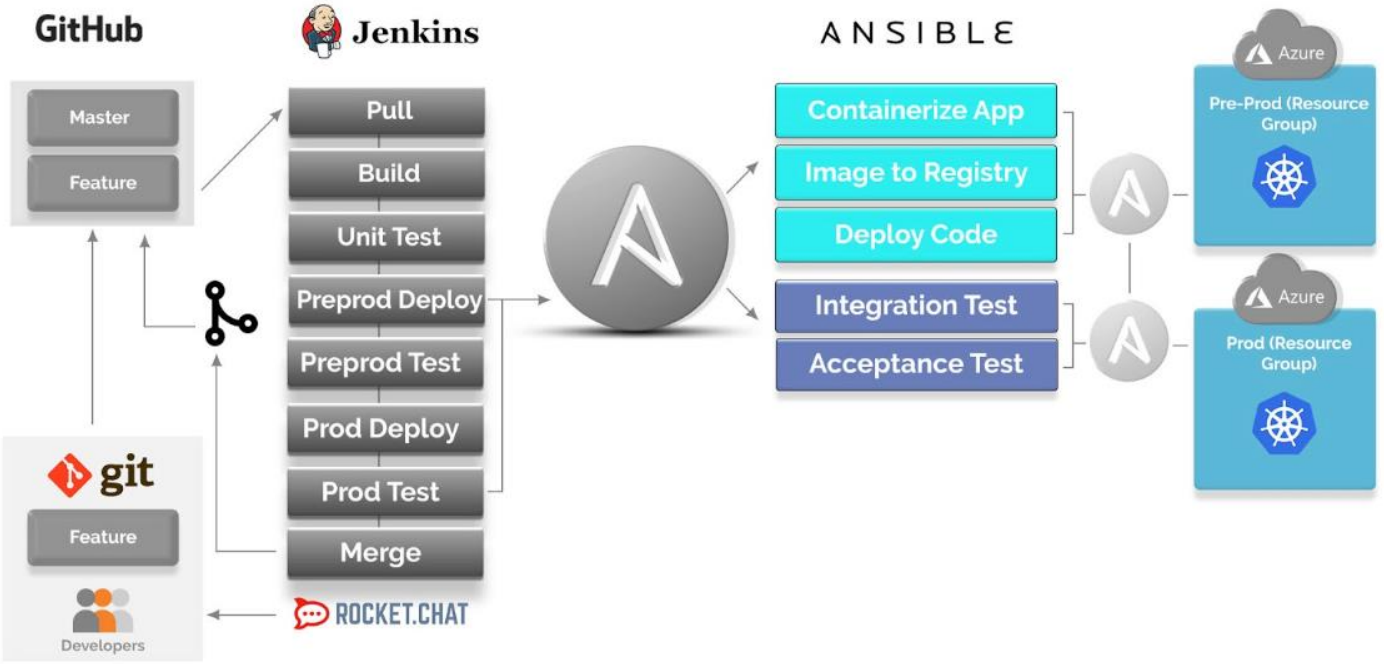


Fig.4. Automating CI/CD workflow using Ansible

Benefits: -

1. Making a CI/CD infrastructure takes hours maybe days. But with ansible you can do it in minutes.
2. No need to keep your infrastructure always running and incurring costs, the infrastructure can be provisioned whenever code is pushed to the repo and cleaned up afterwards.
3. Cost efficient way to maintain a CI/CD pipeline.

Conclusion

CI/CD is integral in software building and deployment: smaller teams, constant changes, fast and real-time feedback, and app deployment. Not only do they provide clear benefits to businesses but also to stakeholders such as product owners, development teams, and end-users, just to name a few.

Every new invention or implementation in the system has to go through much debate to see the pros and cons as well as find benefits. The same applies to CI (continuous integration) and CD (continuous delivery) of the software. While the technology stakeholder may find CI/CD processes as additional tasks, once the CI/CD process stabilizes in an organization, it can lead to big advantages, such as reducing costs and increasing ROI.

In addition, with automation, businesses will be able to invest more time in building better mobile apps.

Therefore, CI/CD methodology facilitates building and improving great apps with faster time to market. Also, having a good automation allows for a more streamlined app development cycle, thus enabling one to get through the feedback cycle quicker and build better, more consistent apps.