INFRA-DEVOPS PROPOSAL

For Pizzamannia

DECEMBER 19, 2016

KUMAR SHANTANU

Table of Contents

Introduction

Purpose

The company is going start a pizzeria and they want to show their online presence by having a website and mobile application for the purpose of digital marketing and supporting online ordering of pizza with useful features like tracking of delivery, order status, online payment etc. This document provides an overview of the system architecture and the process involved from an infrastructure devops stand point. This document proposes hosting the infrastructure either in AWS or using Openshift in companies' own data center.

Audience

This document is intended for technical/infrastructure team at the pizzamania to understand the overall architecture design and architectural decisions that were made during the project's design phase.

Technical Components

Core Components

- **NGINX** - Nginx is one of the most popular webservers currently used around the world. This is known as one of the lightest webservers with additional capability to act as a reverse proxy or load balancer. We will be using nginx to proxy the web requests to proxy to the application server as well as to server some static contents such as js, css, images and videos. We will be using nginx version 1.10.
- **Apache Tomcat** - Tomcat is an open source java servlet container to run java code. All of our backend code to process and manage online orders be written in java and will be deployed in a tomcat container. We will be using tomcat version 7.
- **MariaDB** - MariaDB is an open source relation database server made by the original developers of MySQL. We will be using MariaDB to store data related to customer such as customer's profile, order history etc. We will be using MariaDB version 10.1. While using AWS for infrastructure hosting we will be using RDS service with MariaDB engine
- **JasperReport** - JasperReport is and open source reporting server which provide analytics and reporting. We will be using JasperReport to generate reports which intern will be used by marketing department and also will be used by the offer-generation engine to generate more relevant offers for customers.
- **Solr** - Built on apache lucene, solr is a search platform which provides distributed indexing, replication and load-balanced querying. We will be using solr to primarily store product catalogs.
- **ActiveMQ** – ActiveMQ is and open source messaging broker server. It will be used to send or receive message from one system to another. While using AWS as our infrastructure platform we will be using SQS for this purpose.

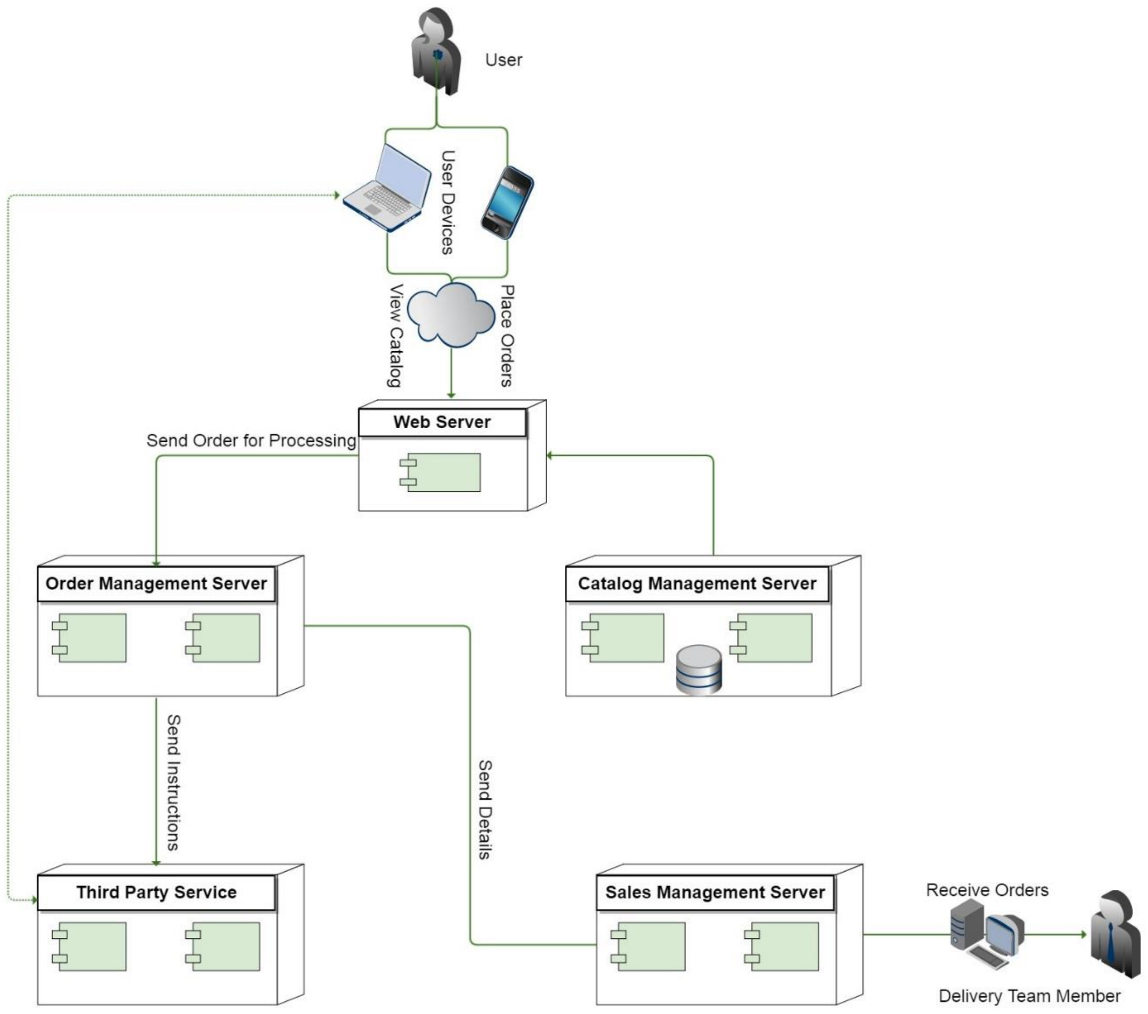
Third Party Components

- GPS Service
- Map Service
- Simple Messaging Service
- Push notification Service

System Architecture

Logical Solution Architecture

The diagram below shows a high level architecture of the technical solution.



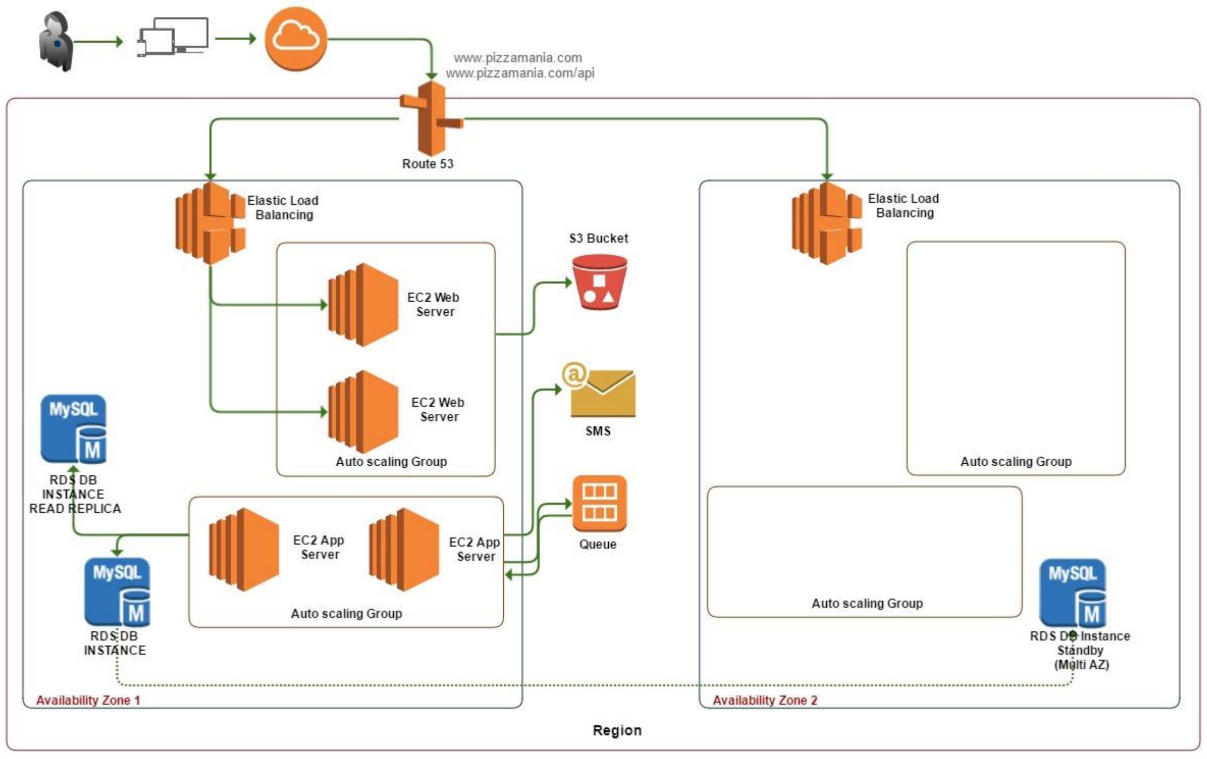
Environments Definition

- **Dev Environment** - This environment is used to perform the sanity tests and will be accessed by Developers, Track Leads and CM.
- **Integration Environment** - This environment is used to perform the functional and systems testing of the application. The environment is owned by QA Manager and builds deployed to this environment are accessed by the QA team.
- **Pre-pod Environment** - Pre-pod environment is configured to mirror production. This will be used for Performance and Security testing
- **Production Environment** - It is a Go-live Environment where the brand sites and new Product releases will be available for End-users.
- **DR Environment** - DR environment is the replica of the PROD environment and will serve the live site in case the PROD environment is not available or goes down.

Physical Architecture

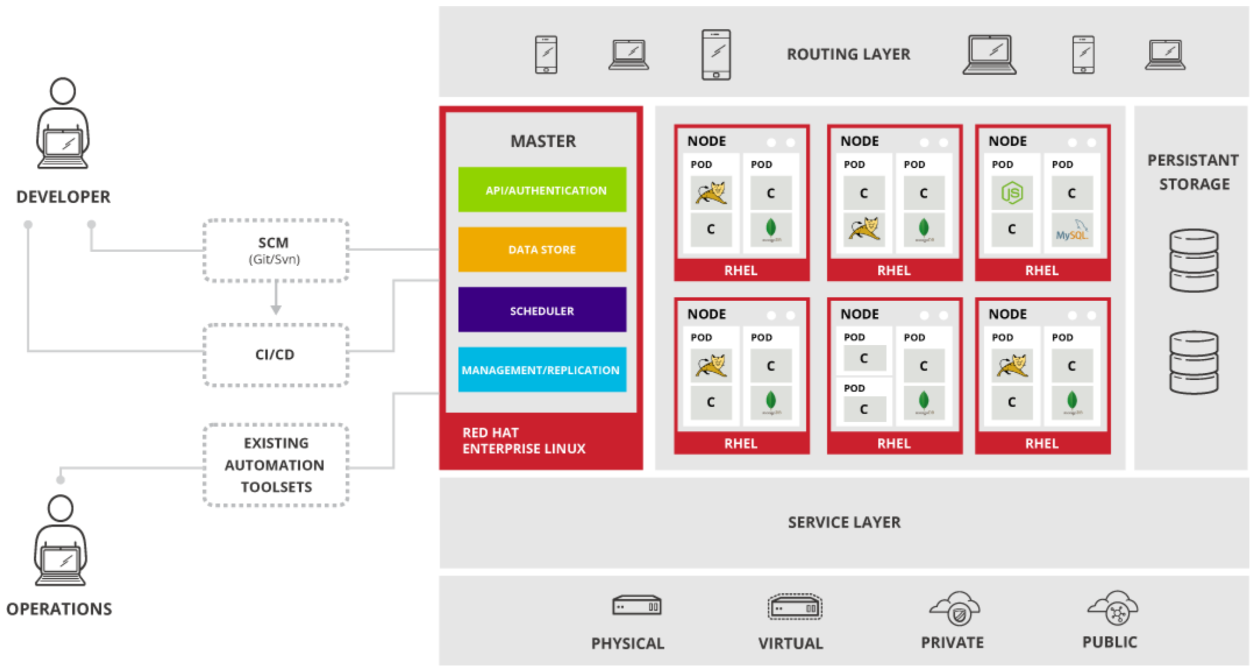
AWS

This physical architecture is designed based on the assumption that we will be using AWS for infrastructure hosting.



On-Prem

This physical architecture is designed based on the assumption that we will be using on-prem infrastructure. This infrastructure will basically host openshift origin as a PAAS provider.



Devops Implementation

Tools Used

- Bitbucket for source code management
- Jira for tracking bugs
- Jenkins 2.0 as a CI tool
- Maven to build java based projects
- Gradle to build android application
- Sonarqube for static code analysis
- Nexus for hosting artifacts
- Selenium for functional testing
- Ansible for configuration management
- Openshift client tool for managing openshift components
- Automated load testing using JMeter
- Automated security testing using OWASP
- Automated unit testing
- Crucible or fisheye for code review
- CSSLint, JSHint, for css/javascript code quality check
- ELK for log management

Infrastructure Provisioning

AWS

While AWS provides a nice web console to provision required infrastructural services, we will be using ansible to automate the provisioning of any AWS services, like ec2, rds or vpc. We will be using python based boto AWS SDK with ansible to automate the provisioning/de-provisioning of AWS services. While provisioning through ansible scripts it will take care of installation and configuration of packages like nginx, tomcat or mariadb. It will also take care of installing and configuring of monitoring agents (nrpe) on to the provisioned machines.

On-Prem

Installation of physical components are out of the scope of works here. We are assuming we will be getting a four node cluster installation of Openshift Origing ready to be used. The top level components like containers, storage, networking etc will be provisioned and maintained by using openshift client tools, bash scripts and Jenkins in conjunction.

Continuous Development & Integration

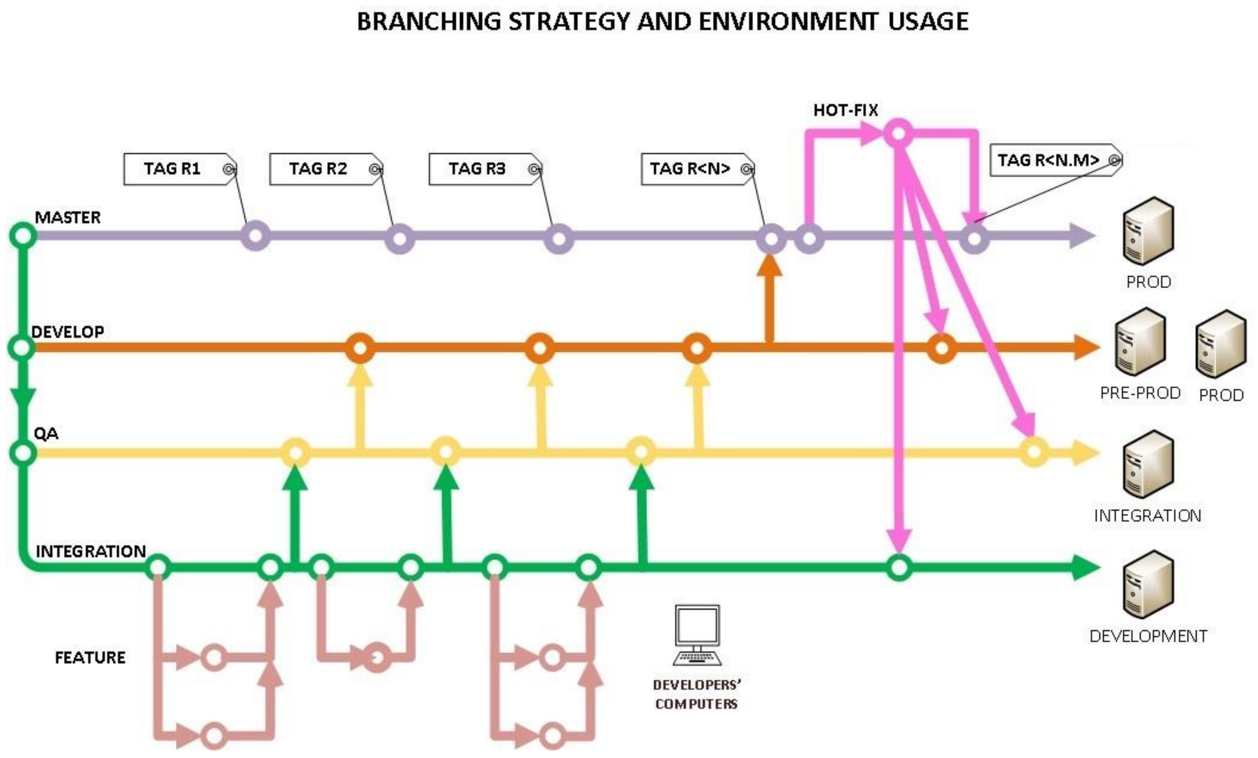
Branching and Merging

OPERATIONAL PROCESS

1. A release specific INTEGRATION branch will be created from master. This branch is to hold all changes meant to go-live with the current release.
2. Feature branches like feature/MGLS-1234, MGLS-5678 etc. for each story being developed will be created by developers from INTEGRATION branch.
3. Once the changes are submitted into these feature branches, developers will create pull requests in Bitbucket requesting at least one reviewer (enforced from within Bitbucket) to review code before the feature branch can be merged in INTEGRATION branch. Reviewer may approve and merge or may reject the request citing proper reason.
4. Continuous Integration setup polls INTEGRATION branch and will build and deploy it's artifacts as soon as a change is pushed in this branch. Deployment off this branch will happen on DEV environment
5. Once the changes for a sprint are accumulated in INTEGRATION branch, this branch is merged into QA branch by raising pull requests and having atleast one reviewer. Automatic builds will happen off QA to deploy onto INT environment
6. For deploying onto Pre-prod environment, code will be merged from QA to develop branch by Build team. Artifacts thus generated will be stored in Nexus with proper versioning and also deployed on Pre-Prod environment.
7. Pts. 3 to 4 repeat till all the stories of a release are delivered on Pre-Prod.

8. If the UAT, which happens on Pre-Prod environment suggests a 'go' for live, artifacts deployed on Pre-Prod are deployed on Prod (no new compilation will be done; artifacts will be pulled off Nexus and deployed on Production). Develop branch will be merged into master and a tag would be created. master branch will thus hold code that is running on Production
9. For the subsequent releases, steps #1 to #7 repeat.
10. For a hot fix, branch will be created from tag from the master branch of the current live codebase, fix be made there and tested on a SELF-DEPLOY environment which would be a replica of Prod and on successful testing will be deployed on Production. This branch will be merged into master, develop, qa and integration and a tag will be created.

Following diagram illustrate the branching strategy.



Branches and their target Environments for RI

S.No.	Branch Type	Branch Name for Release 1	Branch Writable By	Builds Invocation	Target Environment
1	Feature	MGLS-wxyz	Developers	By Developers	Local Workstations
2	Integration	R1-development	Developers, Track Leads/ Reviewers	Automatic via Jenkins	Dev
3	QA	R1-qa	Developers,, Track Leads / Reviewers	Automatic via Jenkins	Integration
4	develop	Develop	Build Manager	Manual via Jenkins	Pre-Prod, Prod
5	master	Master	Build Manager	N/A	N/A
6	hot-fix	hotfix/MGLS-abcd	Track Leads / Reviewers, Build Manager	Manual via Jenkins	Pre-Prod, Prod

Code Versioning Strategy

The code versioning strategy will be based on the semantic bundle versioning. It will have 4 segments, 3 integers and a string respectively named *major.minor.service.qualifier*.

- The major segment will indicate breakage in the API
- The minor segment will indicate "externally visible" changes
- The service segment will indicate bug fixes and the change of development stream
- The qualifier segment will indicate a particular build

Here is an example of what it will look like

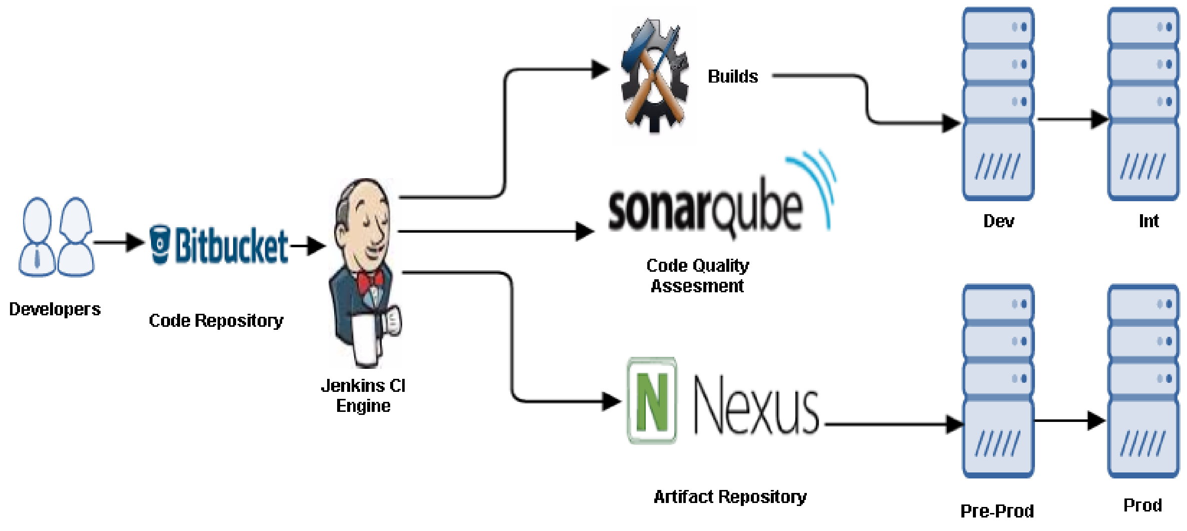
```
First development stream
 - 1.0.0

Second development stream
 - 1.0.100 (indicates a bug fix)
 - 1.1.0 (a new API has been introduced)
 The plug-in ships as 1.1.0

Third development stream
 - 1.1.100 (indicates a bug fix)
 - 2.0.0 (indicates a breaking change)
 The plug-in ships as 2.0.0

Maintenance stream after 1.1.0
 - 1.1.1
 The plug-in ships as 1.1.1
```

Build and Deploy Process – AWS

1. Based on the story assigned to them, developers themselves will create a branch of type "feature" from within the JIRA interface. The feature branches should have the same name as the JIRA id associated with that story and should be sourced from the current release's integration branch.
2. Developers will "git pull" this feature branch in their local workstations and will make the required changes in their local repository.
3. Developers will test their changes off their local feature branch on their workstations and only once they work as expected, the changes should be pushed in the remote feature branch.
4. Developers will then raise a pull request from within Bitbucket to merge the remote feature branch into the integration branch.
5. Reviewer will be notified by email of the pending pull request(s) waiting for his approval. He should review the incoming code and depending upon his findings will either approve and/or merge the pull request or reject it citing appropriate reasons in the Bitbucket interface.
6. Jenkins monitors the integration branch and on finding changes will attempt to build the code; the repository probing is invoked every five minutes. Once the pull request gets merged into the integration branch, Jenkins detects the changes and automatically initiates the Build. If the build is successful, the generated artifact will be automatically deployed on Development environment. If the build fails, email notifications will be sent to the configured addresses.
7. Steps 1-6 repeat till the sprint's code becomes stable. Once that happens, developers will raise a pull request from within Bitbucket to merge the integration branch into the qa branch, mentioning the reviewer in the pull request.
8. Reviewer will be notified by email of the pending pull request(s) waiting for his approval. He should review the incoming code and depending upon his findings will either approve and/or merge the pull request or reject it citing appropriate reasons in the Bitbucket interface.
9. Jenkins monitors the qa branch and on finding changes will attempt to build the code. Once the pull request gets merged into the qa branch, Jenkins detects the changes and automatically initiates the Build. If the build is successful, the generated artifact will be automatically deployed on Integration environment. If the build fails, email notifications will be sent to the configured addresses.

10. Once the sprint(s)' code is ready to be deployed on pre-production, dev leads will raise a request with BnR team for the same. The BnR team will merge the code from qa branch to develop branch and will the build process which will pick code from the develop branch, compile it, store the generated artifact in Nexus and deploy in pre-production environment. This step is repeated till all the stories of a release are deployed on pre-production.
11. After UAT, when the code on pre-production is approved to go on production, the Dev PM / Arch shall make a request to the Build team to do deployment on Production. On receiving such request, the Build team will pick the artifacts already deployed on pre-production environment and will deploy them on Production. Note that there is no re-compilation of artifacts for Production deployment. The code in develop branch is merged in master and tagged with the name of the release.
12. For the next release, the Integration branch is created from the master branch and the version number in POMs, pom.xml etc are increased by one. The above steps are then repeated for the next release.

Following diagram illustrate the high level build process when AWS infrastructure used.

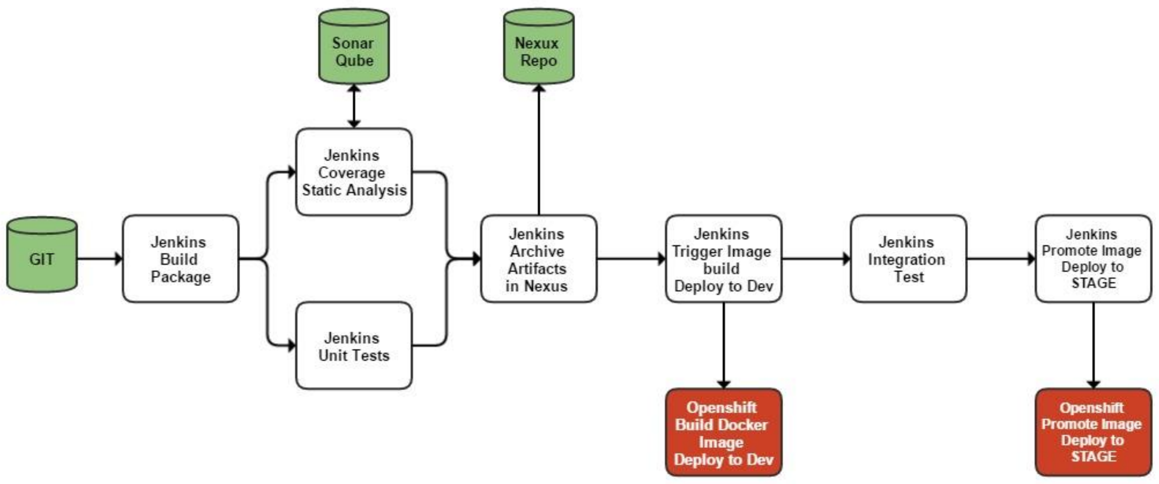


Build and Deploy Process – Openshift

1. Once the remote feature branch is merged into staging, Jenkins clones the code from Bitbucket server built, tested and analyzed for bugs and anti-patterns
2. The WAR artifact build in the previous step is archived in Nexus Repository

3. OpenShift takes the WAR artefact from Nexus Repository and builds a Docker image by layering the WAR file on top of Tomcat 7
4. The Docker image is deployed in a fresh new container in DEV environment
5. A set of automated tests run against the application container in DEV environment.
6. If tests successful, the Docker image is tagged with the application version and gets promoted to the STAGE environment
7. The tagged image is deployed in a fresh new container in the STAGE environment
8. Pipelines pauses at Deploy STAGE for approval in order to promote the build to the STAGE environment. Click on this step on the pipeline and then Promote.
9. QA team will get notification once the stage environment is ready and they can perform UAT. They will be getting a link to the Jenkins pages where they can approve or deny the promotion of code to higher environment.

Following diagram illustrate the high level build process when Openshift infrastructure used.


Sonar Qube
Nexus Repo
Jenkins Coverage Static Analysis
GIT
Jenkins Build Package
Jenkins Archive Artifacts in Nexus
Jenkins Trigger Image build Deploy to Dev
Jenkins Integration Test
Jenkins Promote Image Deploy to STAGE
Jenkins Unit Tests
Openshift Build Docker Image Deploy to Dev
Openshift Promote Image Deploy to STAGE

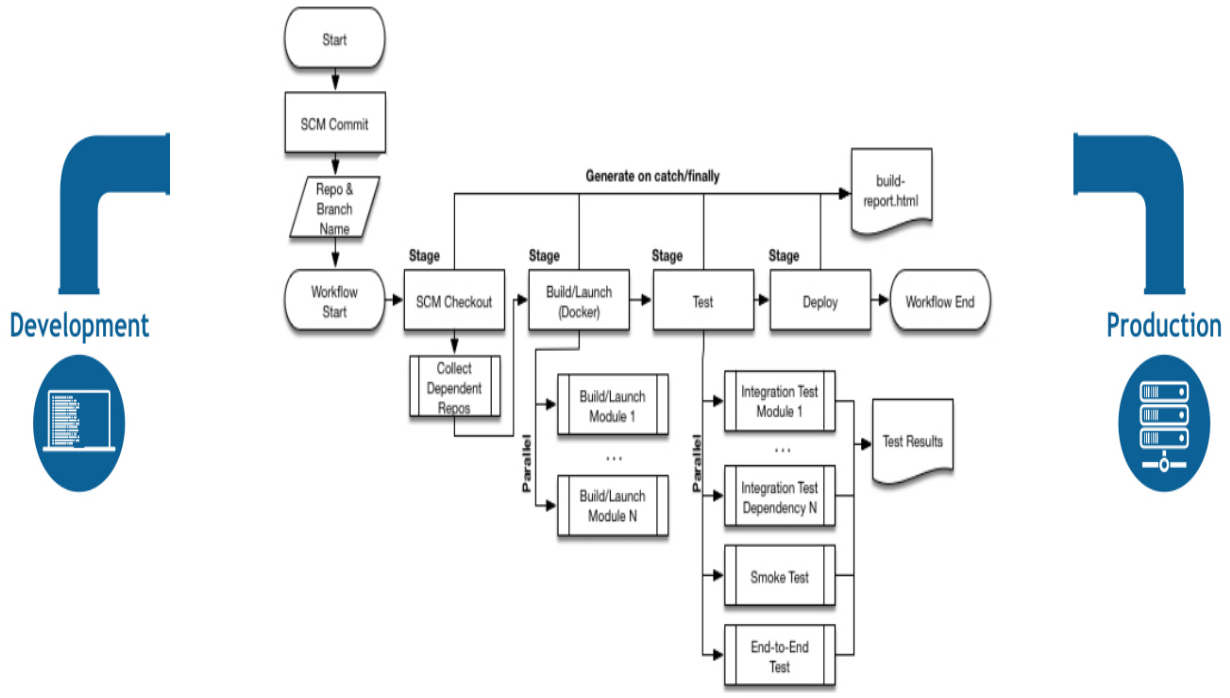
Jenkins and pipeline

Jenkins Pipeline is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins. . Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive continuous delivery pipelines. By modeling a series of related tasks, we will take advantage of the many features of Pipeline like below,

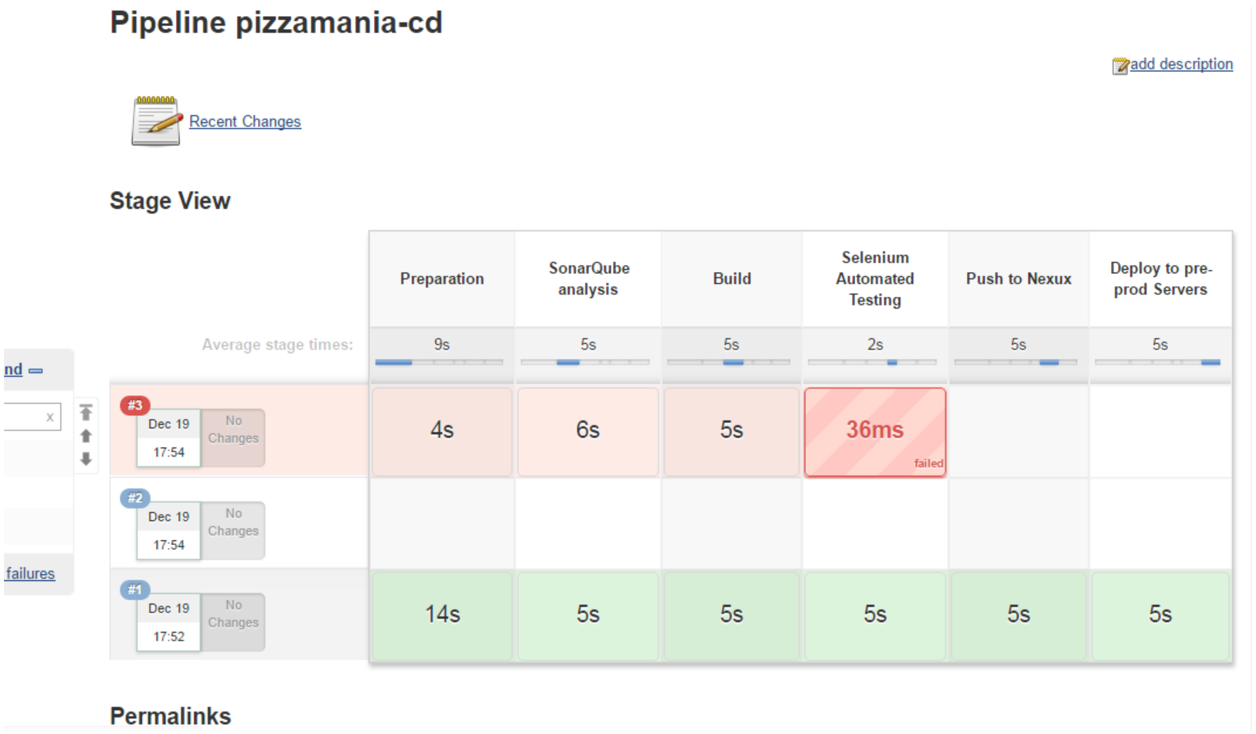
- Code: Pipelines are implemented in code and typically checked into source control, will give teams the ability to edit, review, and iterate upon their delivery pipeline.

- Durable: Pipelines can survive both planned and unplanned restarts of the Jenkins master.
- Pausable: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
- Versatile: Pipelines support complex real-world continuous delivery requirements, including the ability to fork/join, loop, and perform work in parallel.
- Extensible: The Pipeline plugin supports custom extensions to its DSL [1] and multiple options for integration with other plugins.

An example of one continuous delivery scenario easily modeled in Jenkins Pipeline



The flowchart below is an example of one continuous delivery scenario during pizzamania dev process



Pipeline pizzamania-cd

add description

Recent Changes

Stage View

	Preparation	SonarQube analysis	Build	Selenium Automated Testing	Push to Nexux	Deploy to pre-prod Servers
Average stage times:	9s	5s	5s	2s	5s	5s
#3 Dec 19 17:54 No Changes	4s	6s	5s	36ms failed		
#2 Dec 19 17:54 No Changes						
#1 Dec 19 17:52 No Changes	14s	5s	5s	5s	5s	5s

Permalinks

Deployment Strategy

We will be following blue green deployment strategy while deploying artifacts on production and preproduction servers. The blue-green deployment approach will ensure that we have two production environments, as identical as possible. At any time one of them, let's say blue for the example, is live. As we prepare a new release of our software we do your final stage of testing in the green environment. Once the software is working in the green environment, we switch the router so that all incoming requests go to the green environment - the blue one is now idle. Blue-green deployment will also give us a rapid way to rollback - if anything goes wrong we switch the router back to our blue environment.

Non Functional Requirement

High Availability

To make the system highly available at any point of time and support a SLA of 99.99 we have designed all the components of the system to be independently highly available.

- DNS high availability

- **Web server high availability** - Multiple instance of web servers will be running and handling connections individually. The web server configuration will make sure the request always gets forwarded to the correct application server.
- **App server High availability** - To make tomcat application server highly available we will be configuring tomcat cluster with load balancing and session replication. With built-in support for both synchronous and asynchronous in-memory and external session replication, cluster segmentation, and compatibility with all common load balancing solutions, Tomcat servers are ready for the cluster right out of the box.
- **Database server high availability** - We will be using galera cluster to achieve highly available database services. Galera Cluster for MySQL is a true Multimaster Cluster based on synchronous replication. Galera Cluster is an easy-to-use, high-availability solution, which provides high system uptime, no data loss and scalability for future growth. It provides synchronous replication with active-active multi-master topology. It allows applications to read and write to any cluster node.

Scalability

Manual Scaling

- Aws Environment

Manual scaling will be achieved by using ansible scripts whenever needed.

- Openshift Environment

Manual scaling will be achieved by using oc client tools.

Auto Scaling

- **Aws Environment** - To achieve auto scaling in AWS environment we will

    - Define the AMI instance and create an Auto Scaling Group to launch instances into.
    - Use Cloud Watch to monitor the instance(s), and when certain configurable events happen, you can launch more instances based on the AMI template defined.
    - EC2 instances launch behind the Elastic Load Balancer (ELB) you define (External and internal).
    - The ELB will send traffic in a round-robin pattern between all the instances assigned to it, and can control in real time how many instances you want to launch to cover sporadic bursts of high-volume traffic, and keep the least defined instances running during traffic lulls. If any of the EC2 instances fails to respond, the ELB will detect it and launch a replacement
    - Cloud Watch lets you configure alarms that trigger auto scaling policies to launch additional EC2 instances into your auto scaling group when network traffic, server load, or other measurable statistic, gets too high example, 80% usage. Each server is a duplicate instance of the AMI you define in your auto scaling configuration.
    - The ELB can automatically spreads out the incoming visitors between all the servers in the defined Auto scaling Group. You can set a minimum and maximum number of

instances in your group. Auto scaling Group can also be configured to decrease the number of instances to the minimum defined when network traffic drops below, say 20%.

More details can be found at http://docs.aws.amazon.com/autoscaling/latest/userguide/GettingStartedTutorial.html

- **Openshift Environment** - We will create a horizontal pod autoscaler with the oc autoscale command or using yaml files. There we will specify the minimum and maximum number of pods we want to run, as well as the CPU utilization our pods should target. After a horizontal pod autoscaler is created, it will attempt to query Heapster for metrics on the pods. After metrics are available in Heapster, the horizontal pod autoscaler computes the ratio of the current metric utilization with the desired metric utilization, and scales up or down accordingly. The scaling will occur at a regular interval, but it may take one to two minutes before metrics make their way into Heapster.

Disaster Recovery

Disaster recovery solution is out the scope of this document.

Security

To make the application secure we have identified these areas where the risks may be located.

- Vulnerabilities in networks
- Vulnerabilities in design
- Vulnerabilities in systems
- Vulnerabilities in applications
- Vulnerabilities in database
- External threats
- Internal threats

Solution to these problems on a very high level are listed here.

- Separation of Networks
  - No direct traffic will be allowed from external to internal networks.
  - All that is not explicitly allowed will be denied.
- Isolated Network (DMZ)
  - Buffer network between external and internal networks.

  - Production and test networks will be separated.
- Sterile Environment
  - DMZ systems will treated as unsafe
  - Will practice good housekeeping (debugs, dumps, temporary files…)
- No RA to Mgmt Systems
  - Applications won't offer administrative controls directly via presentation layer (port, url, etc)
  - Network or server remote access will be blocked (ssh, telnet, pca, rdp, etc)
  - Will be allowed only via authenticated, secure RA services (IPSec, VPN, SSL-VPN)
- Lockdown before exposure
  - Require Dev, QA, SA, Sec to complete testing from documented processes
  - Require an approval process to place system in DMZ
  - Accept "residual risk" and authorize system operation in writing
- Least Access
  - Access granted at the minimum level required
  - Practice at system and network level
- Least Use
  - Servers will be used only for one purpose
  - Unused services and applications will be removed
- Monitoring
  - Will be using IDS/IDP
  - Will be offloading logs to central repository
  - Custom apps will generate logs
  - Will keep understanding what's going on - situational awareness
- Encrypted Data Transfer
  - Will use standard encryption protocols for data in transit (SSL/TLS)
  - End-to-end encryption will be used.(transit to rest)
- Application Security
  - Session Management will be done
  - Data and InPut Validation will be done
  - Code will be checked for any Cross Site Scripting vulnerability
  - Command Injection Flaw will be checked.
- Unique to Database
  - No users will access the database
  - Developers will not access production
  - Remove access to system will be only via stored procedures
  - All access to data will be only via stored procedure
  - There won't be any code in the tables

Monitoring

Below are the list of different tools will be using for different purposes.

Nagios - Nagios is a free and open source computer-software application that monitors systems, networks and infrastructure. Nagios offers monitoring and alerting services for servers, switches, applications and services. It alerts users when things go wrong and alerts them a second time when the problem has been resolved.

AppDynamics – AppDynamics an application performance management tool which ensures application performance and user satisfaction by proactively monitoring applications end-to-end. It monitors every transaction but intelligently capture details of only the anomalous transactions, making the platform scale to meet the demands of large enterprises

Thank You