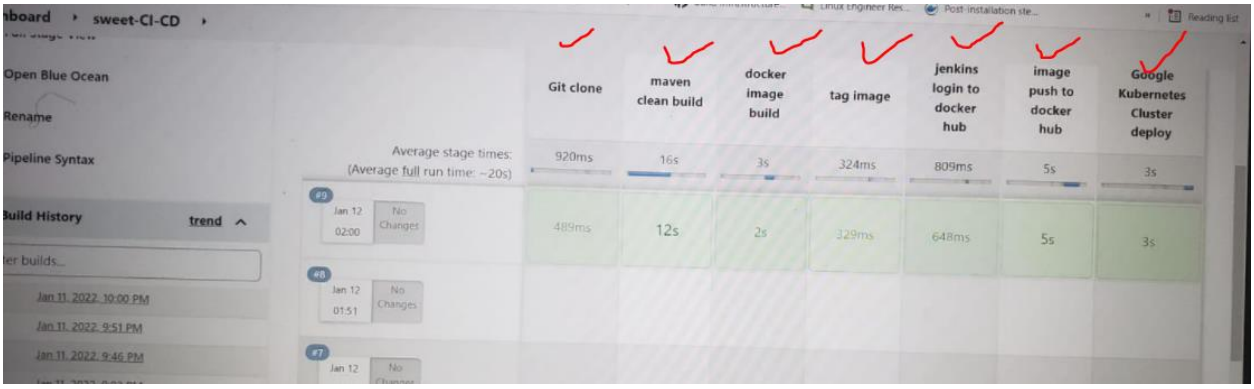
FINAL NOTES – CI/CD PIPELINE

If you are like me, you probably get confused about the difference between **Continuous Delivery and Continuous Deployment** and how they relate to **Continuous Integration**.

The current conventions give us these approximate definitions:

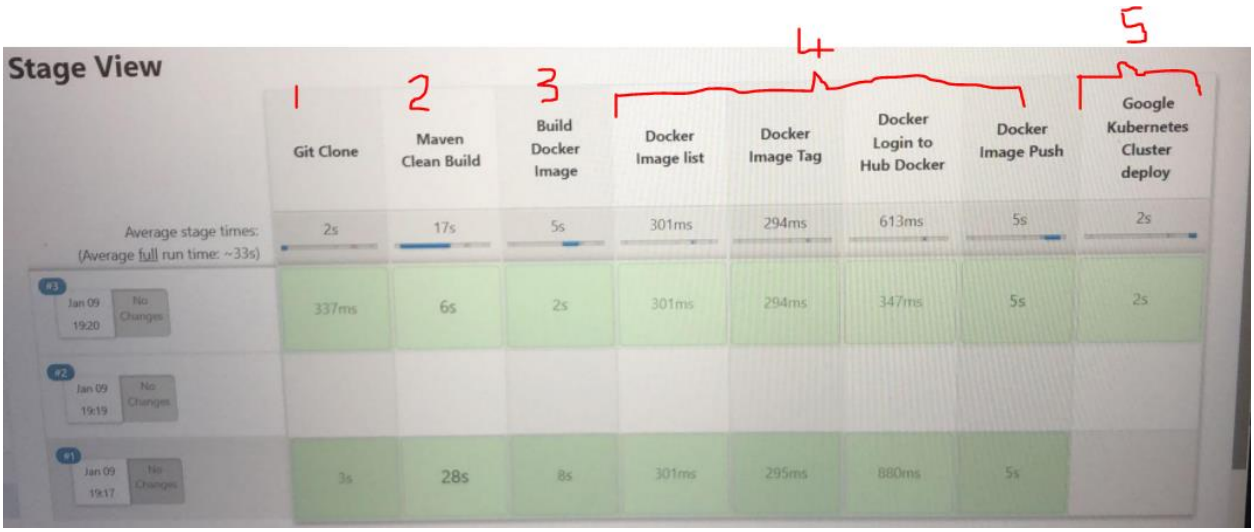
- Continuous Integration - Every commit gets tested, and retested.
- Continuous Delivery - Every commit gets tested and, if successful, is turned into a release/image application that may be deployed to staging or live production**.**
- There are two types of environments in the production – **staging and live**
- Continuous Deployment - Every commit gets tested and, if successful, is turned into a release/build/application that always gets **deployed to live production (Kubernetes cluster)**

Examples of Images from the class CI/CD Pipelines



Example of successful automation from git cloning with Jenkins git:Git to software/application to live production/Deployment in Google Kubernetes Engine/cluster.

This is automation in every stage that many companies are not comfortable with because simple mistakes can pass through, affecting the image of the company heavily.



Continuous Integration Stages - 1, 2 and 3

Stage 1,2,3 is part of the continuous integration - Ultimately, CI ends when a build completes successful initial testing and is ready to move to more comprehensive testing.

Preparation might include packaging the build into a deployable image, such as a container or virtual machine (VM) image, before making it available to dedicated testers.

In our class, we are packaging the build into a docker image ready for delivery in the staging area in the production.

Before Initiating Stage 4 (Continuous delivery stage)

We must initiate **a merge request** to deliver the **build docker imag**e from **QAs** environment to **a repository** located in the **Staging area** in the production environment.

Continuous Delivery Stage - 4

Stage 4 – Merge Requested is created using service now

Application is delivered to the staging area of production when the image is pushed to the production repository (docker hub or Amazon Elastic Container Registry)

Activities in this stage include:

Advanced testing, which include functional (is the build/application working), user acceptance (UAT) (how will the user think about this application), configuration

(more installation of important updates to the application) , and load testing (the ability of the application to handle the traffic of the users without throwing errors).

Again, small incremental iterations ensure that any problems revealed in testing are identified and remediated quickly and less expensively than traditional software development approaches.

Continuous Deployment Stage – 5

After Stage 4 advanced testing validates that the build/application meets requirements and is ready for use in **a live production environment, the deployment stage is initiated. The application is deployed to the google Kubernetes cluster.**

More Notes to better your understanding of the CI/CD Pipeline

- CI/CD is a solution to the **problems integrating new code** can cause for development and operations teams (**AKA "integration hell"**).
- Too many branches of the application with new changes that will conflict with each other and slow the entire process of developing a software.
- Integrating new codes with the old ones can always cause the build to break, which can be a hell to deal with.
- This problem is solved by integrating Jenkins, the most famous continuous integration tool with build automation tools, Maven and Ant.
- Maven is not just used as a build tool but can also be used to manage the entire java project because it has many plugins downloaded from the central location, which can be used to display the logs of the entire system.
- Specifically, CI/**CD introduces ongoing automation** and **continuous monitoring throughout the lifecycle** of apps, from integration and testing phases to delivery and deployment.
- Continuous integration (CI) focuses on the early stages of a software development pipeline where the code is built and undergoes initial testing.
- Multiple developers simultaneously work on the same codebase and make frequent commits to the code repository.
- Build frequency can be daily or even several times per day at some points in the project's lifecycle.

- These small, frequent builds enable easy and low-risk experimentation and the ability to roll back or abandon undesirable outcomes easily.
- CI employs a variety of tools and automation techniques to create builds and shepherd them through initial testing, such as sniff or unit testing, along with more comprehensive integration testing.
- CI is also noted for its rapid and detailed feedback.
- The limited nature of each iteration means that bugs are identified, located, reported, and corrected with relative ease.
- Ultimately, CI ends when a build successfully completes initial testing and moves to more comprehensive testing.
- Preparation might include packaging the build into a deployable image, such as a container or virtual machine (VM) image, before making it available to dedicated testers.

The QA

- QAs do the screening of quality of the code and builds
- Helping the developers to come up with better ways of developing and testing the source code
- QAs helps devs gain a better understanding of testing philosophies and processes in the company
- Developers have not been trained to look for quality issues, only fix them.
- Even when they care about doing so, they will probably miss issues.
- In DevOps, QA responsibilities start with imparting knowledge and training to developers.

The "CD" in CI/CD refers to **continuous delivery** and/or **continuous deployment** which are related concepts that are sometimes used interchangeably. Both are about automating further stages of the pipeline, but they're used separately to illustrate how much automation is happening.

Continuous delivery (CD) picks up where CI leaves off. It focuses on the later stages of a pipeline, where a completed build is thoroughly **tested, validated, and delivered for deployment**.

Continuous delivery can -- but does not necessarily -- deploy a successfully tested and validated build.

Staging environment of the production

CD likewise relies heavily on tools and automation to take a build through **advanced testing, including functional, user acceptance, configuration, and load testing.**

These validate that the build meets requirements and is ready for use in a production environment.

Again, small incremental iterations ensure that any problems revealed in testing are identified and remediated quickly and less expensively than traditional software development approaches.

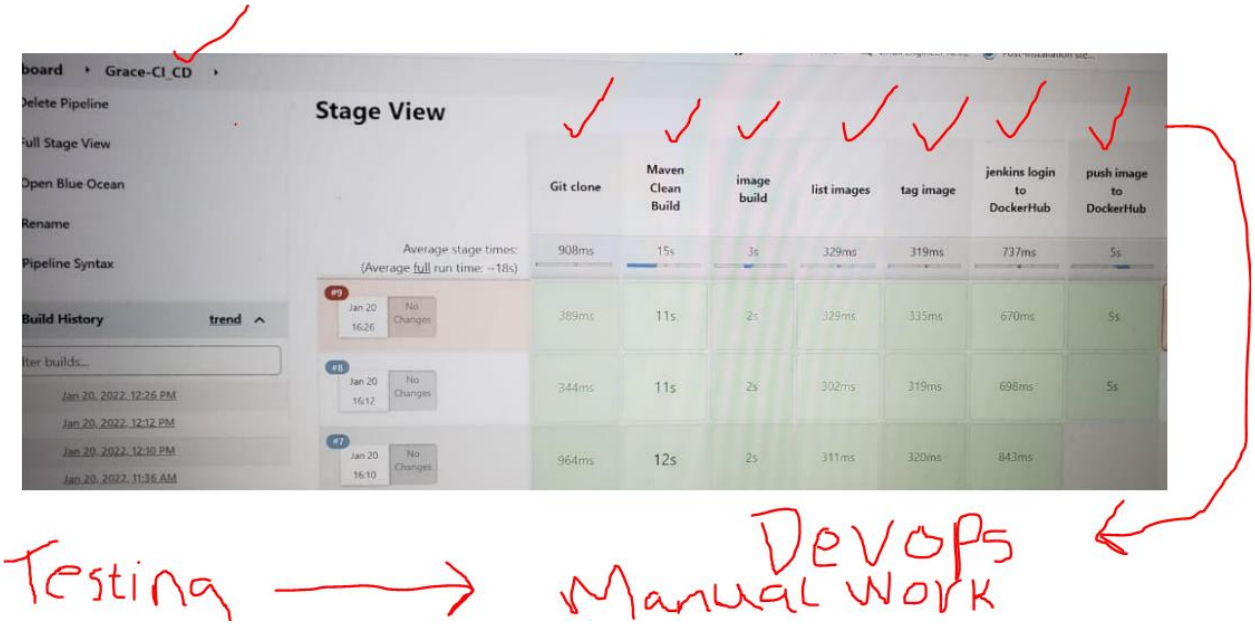
Live production – Deployment to the end-users

Continuous deployment

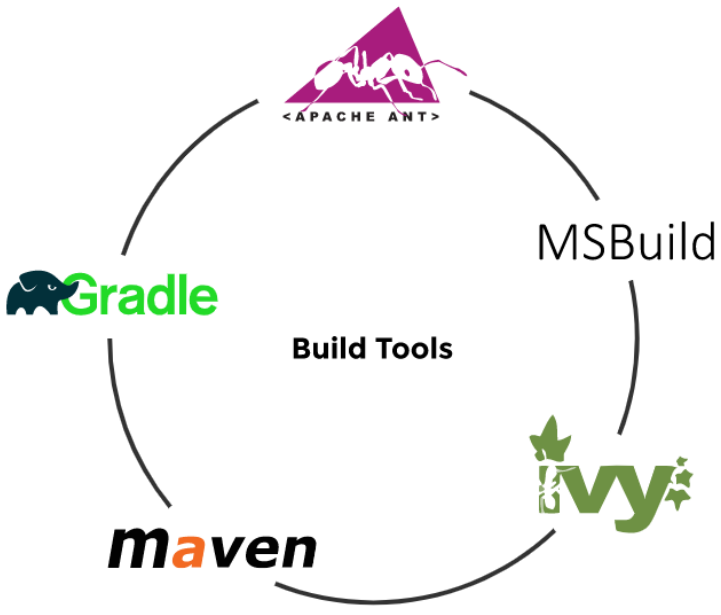
Continuous deployment (also CD) follows the same basic steps as continuous delivery.

The principal difference between delivery and deployment is that continuous deployment automatically deploys each validated application/build **to production**.

Very Important



By comparison, continuous delivery typically stages the validated build for manual deployment or other human authorization. This human intervention starts when the application is delivered to the staging area in the production.



Different Build Automation Tool – We have used 2 in class

Build

In software development, a **build is the process of converting source code files into standalone software artifact(s) that can be run on a computer**. The resulting artifact is sometimes referred to as the build. In an organization, there are usually several branches of code with varying degrees of stability that result in nightly builds and get deployed in various environments such as stage, test, or production.

Build tool

The **process of building a computer program is usually managed by a build tool, a program that coordinates and controls other programs**. Examples of such programs are Make, Gradle, Meister by OpenMake Software, Ant, Maven, Rake, SCons, and Phing. The build utility typically needs to compile the various files in the correct order, download dependencies, run unit and integration tests, and any other related tasks for producing binaries that can be deployed.

Artifact

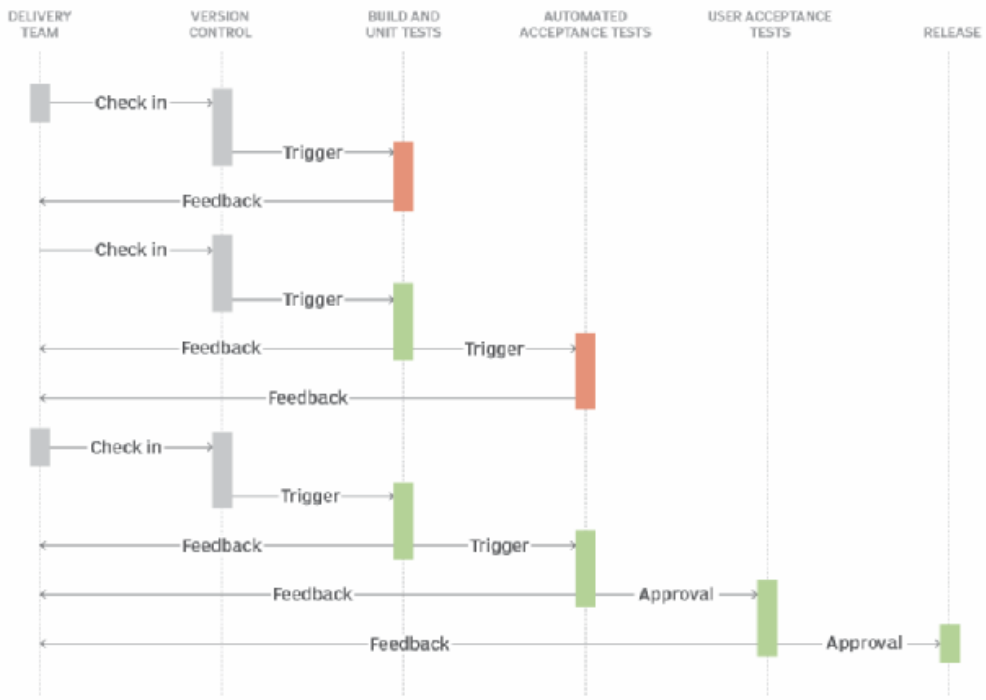
The word artifact is from the Latin phrase **arte factum**: arte meaning **skill** and factum meaning **to make**. In the software development life cycle, artifact usually refers to *objects* that are produced by people involved in the process. Examples would be design documents, data models, workflow diagrams, test matrices and plans, setup scripts including compiled binaries, and software packages. In most software development cycles, there's usually a list of specific required artifacts that someone must produce and put on a shared drive or document repository for other people to view and share. Artifact may occasionally refer to the released code (in the case of a code library) or the released executable (in the case of a program). An artifact is the byproduct of software development.

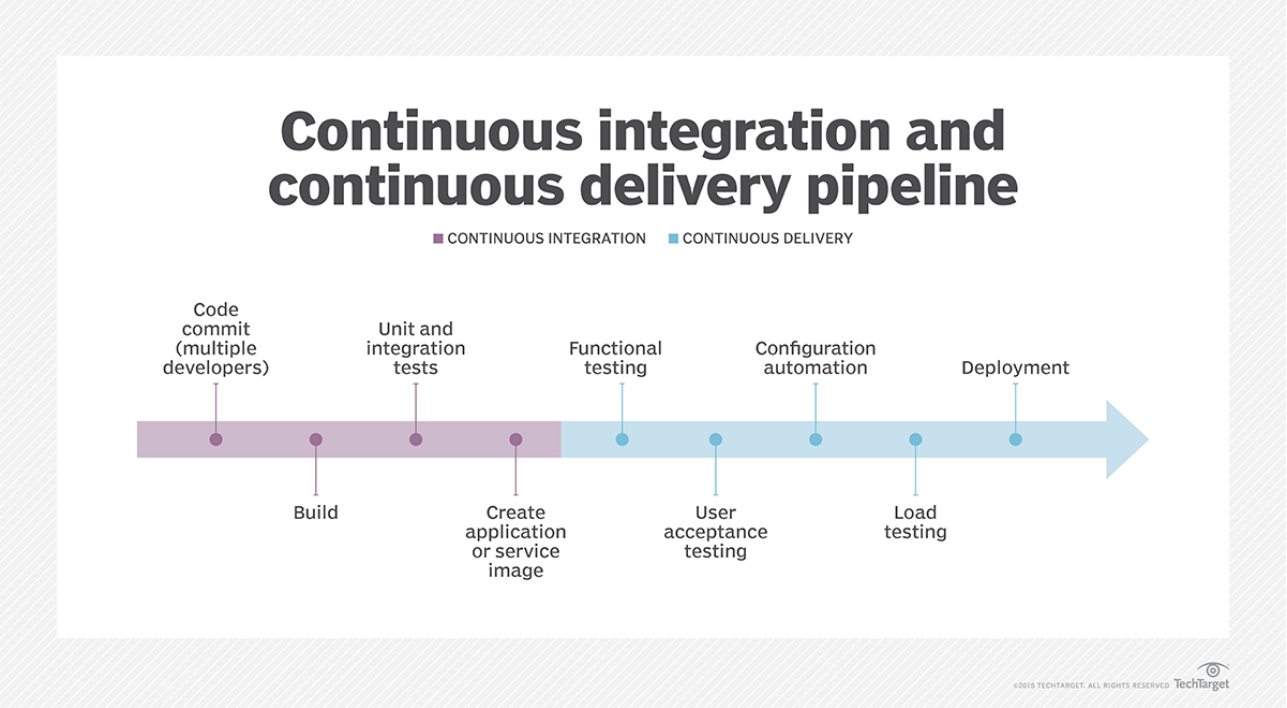
Build automation

Build automation is **the process of automating the creation of a software build and the associated processes. This includes compiling computer source code into binary code, packaging binary code, and running automated tests**. A build - the process that converts files and other assets under the developers' responsibility into a software product in its final or consumable form - is automated when a final artifact(s) can be produced without direct human intervention and can be performed at any time with no information other than what is stored in the source code control repository.

More Images

Continuous delivery process





The benefits of a CI/CD pipeline include the following:

- **Efficient software development.** Smaller iterations (steps) allow for easier and more efficient testing. The limited scope of code in each new iteration and the scope to test it make it easier to find and fix bugs. Features are more readily evaluated for usefulness and user acceptance, and less useful features are easily adjusted or even abandoned before further development is wasted.
- **Competitive software products.** Traditional software development approaches can take months or years, and formalized specifications and requirements are not well suited to changing user needs and expectations. CI/CD development readily adapts to new and changing requirements, enabling developers to implement changes in subsequent iterations. Products developed with CI/CD can reach the market faster and more successfully.
- **Freedom to fail.** CI/CD's rapid cyclicality enables developers to experiment with innovative coding styles and algorithms with far less risk than traditional software development paradigms. If an experiment does not work out, it probably will not ever see production and can be undone in the next rapid iteration. The potential for competitive innovation is a powerful driver for organizations to use CI/CD.

- **Better software maintenance.** Bugs can take weeks or months to fix in traditional software development, but the constant flow of a CI/CD pipeline makes it easier to address and fix bugs faster and with better confidence. The product is more stable and reliable over time.
- **Better operations support.** Regular software releases keep operations staff in tune with the software's requirements and monitoring needs. Administrators can better deploy software updates and handle rollbacks with fewer deployment errors and needless troubleshooting. Similarly, IT automation technologies can help speed deployments while reducing setup or configuration errors.
- **Better communication along the entire SDLC**
- **Better team coordination and cooperation along the entire SDLC**
- **Speed**
- **Consistency**
- **Automation**
- **Integrated feedback loops**
- Security – QAs and other integrated and advanced testing help to reduce the vulnerability of the software

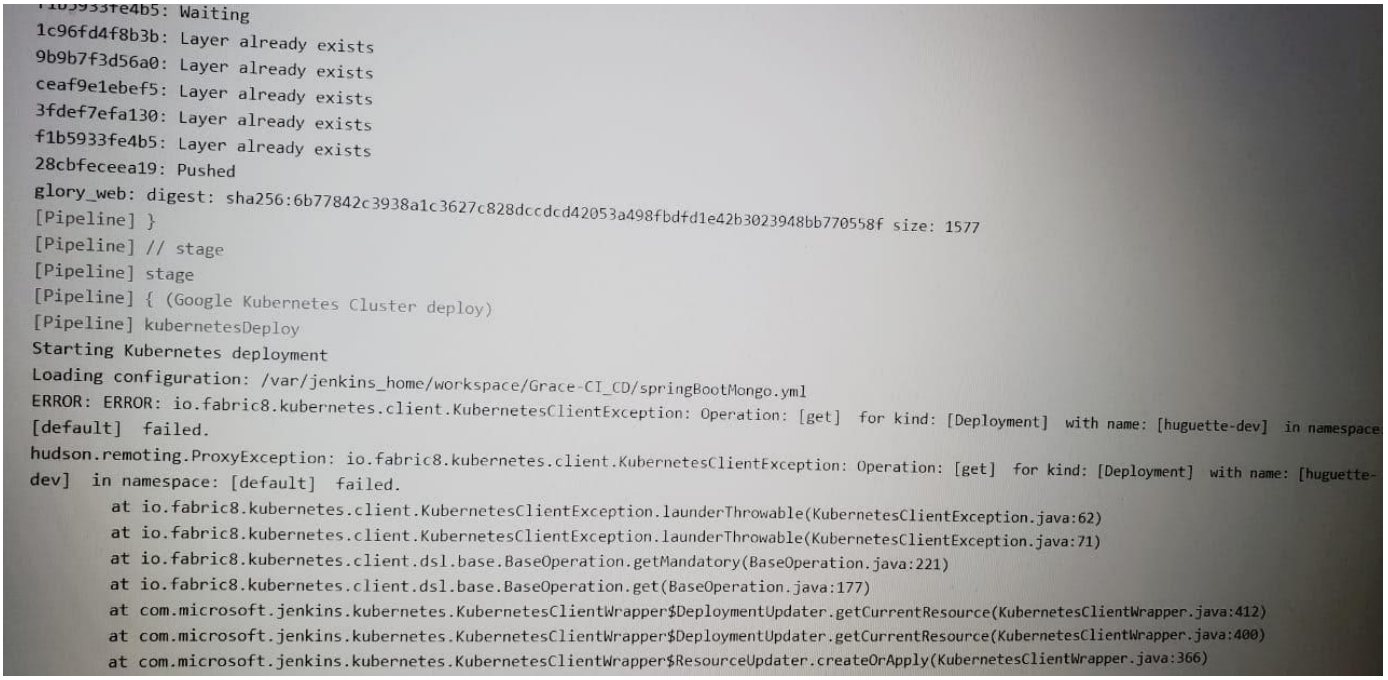
Disadvantages of CI/CD pipeline

Heavy investment in the team – The pipeline requires hiring human resources with deep knowledge and skills.

Not every company is interested in automating the entire process from source code push to deployment/ live production.

Do not for get Logs and their importance

Example of logs


f1b5933fe4b5: Waiting
1c96fd4f8b3b: Layer already exists
9b9b7f3d56a0: Layer already exists
ceaf9e1ebef5: Layer already exists
3fdef7efa130: Layer already exists
f1b5933fe4b5: Layer already exists
28cbfeceea19: Pushed
glory_web: digest: sha256:6b77842c3938a1c3627c828dccdcd42053a498fbdfd1e42b3023948bb770558f size: 1577
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Google Kubernetes Cluster deploy)
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/Grace-CI_CD/springBootMongo.yml
ERROR: ERROR: io.fabric8.kubernetes.client.KubernetesClientException: Operation: [get] for kind: [Deployment] with name: [huguette-dev] in namespace
[default] failed.
hudson.remoting.ProxyException: io.fabric8.kubernetes.client.KubernetesClientException: Operation: [get] for kind: [Deployment] with name: [huguette-
dev] in namespace: [default] failed.
at io.fabric8.kubernetes.client.KubernetesClientException.launderThrowable(KubernetesClientException.java:62)
at io.fabric8.kubernetes.client.KubernetesClientException.launderThrowable(KubernetesClientException.java:71)
at io.fabric8.kubernetes.client.dsl.base.BaseOperation.getMandatory(BaseOperation.java:221)
at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.java:177)
at com.microsoft.jenkins.kubernetes.KubernetesClientWrapper$DeploymentUpdater.getCurrentResource(KubernetesClientWrapper.java:412)
at com.microsoft.jenkins.kubernetes.KubernetesClientWrapper$DeploymentUpdater.getCurrentResource(KubernetesClientWrapper.java:400)
at com.microsoft.jenkins.kubernetes.KubernetesClientWrapper$ResourceUpdater.createOrApply(KubernetesClientWrapper.java:366)