# SecDevOps - Automated CI/CD Pipeline for Elite

Authors : **Payal Moondra** (Senior SDET, HackerEarth), **Karthik R** (SDET, HackerEarth)

**ABSTRACT**

Every fast paced work environment requires a high level of efficacy throughout the process and primarily in a startup, creating an impact everyday is required for them to sustain in the market. It is necessary to have features rolled out in regular cadence and some features/fixes are on demand and tend to have high release frequency. In these scenarios, the deployment is done everyday, where all the features are pushed to production. To make sure that the new features don't conflict or impact the existing ones, it is also important to have a well-devised pipeline that cuts off manual efforts of testing and debugging for the feature to be successfully pushed to production.

We will be discussing how our improved pipeline will help the QA community to reduce manual efforts, early detection of bugs and also how to enhance the code quality early in the development phase.

**AUDIENCE**

Technology areas - SecDevOps, CI/CD pipeline, Deployment, Infrastructure. The Audience include and are not limited to DevOps Engineers, QA Engineers, Developers and Build & Release Engineers

**INTRODUCTION**

A Continuous Delivery pipeline helps you automate steps in your software delivery process, such as initiating code builds, running automated tests, and deploying to a staging or production environment. Automated pipelines remove manual errors, provide standardized development feedback loops and enable fast product iterations. In this paper we will be discussing in detail about each component of our pipeline and how our pipeline would create an impact for fast moving organisations.

Four factors are important while benchmarking yourself with the elite performers in the DevOps industry.

a. **Lead time for change** (< 1 day) - For the primary application or service you work on, what is the lead time for changes (i.e., how long does it take from code committed to code successfully running in production)
b. **Release frequency** (daily) - For the primary application or service you work on, how often does an organization deploy code to production or release it to end users
c. **Time to recovery** (< 1 hour) - For the primary application or service you work on, how long does it generally take to restore service when a service incident or a defect that impacts users occurs (e.g., unplanned outage or service impairment)
d. **Change failure rate** (0%-15%) - For the primary application or service you work on, what percentage of changes to production (e.g., lead to service impairment or service outage) and subsequently require remediation (e.g., require a hotfix, rollback, fix forward, patch)

To achieve all these factors, it is important to have a well organised development and deployment practices where QA acts as a bridge for the same to happen seamlessly.

**PROBLEM STATEMENT**

We were medium performers in 2019 with a deployment where fault isolation was cumbersome and slower. Even the smaller code

changes were difficult and had more unintended consequences. Release cycles were longer and this blocked more features to be released that create impact in the business. Some of the notable metrics from our archives for defects are as follows (Note: these are P0 and P1 defects before the implementation of the pipeline)

- Q2'19 - **93**
- Q3'19 - **69**

In addition to the above, each deployment cycle consumed around 3 hours with various manual interventions and this inturn required QA engineers to validate the feature in production manually where the security validation and code quality goes for a toss.

## APPROACH & IMPLEMENTATION

After careful consideration of our infrastructure and requirements, we are using Jenkins for Continuous Integration and Continuous Delivery. We are using Jenkins declarative pipeline to orchestrate the deployment process. To achieve the continuous integration and delivery, below is the process :

1. The developer checks in the code into the feature branch and creates a PR to develop.
2. Pylint is run on this PR and confirms if the PR is good to go to the next stage based on the configuration(.pylintrc).
3. Then the sonarqube analysis is run on the code to check any flaws, vulnerabilities.
4. Once the pylint and sonar output is green, it proceeds to the next stage and starts running integration tests on PR.
5. If the integration tests pass on PR, the feature branch gets merged to develop.
6. The develop branch gets deployed to staging and the functional smoke tests will now run on the development before it gets delivered to QA for sign off.
7. Once signed off, the PR to release from develop gets created. All the above checks run on this PR before getting merged to release.
8. Once done, the release gets deployed to the preprod (staging) and automated functional tests run on preprod.
9. On preprod, the security tests and performance tests will run.
10. If all of the tests above pass, the pipeline is paused until QA manually approves it. Once approved, the release finally gets deployed on production.

## STAGES OF PIPELINE

### Static Code Analysis

Catching flaws earlier can save a lot of failures that may occur in building the code or even in integration tests. This, in turn makes the deployment process faster. To find flaws in the code using static code analysis over dynamic also reduces the cost of fixing the bugs.

**i) Pylint:** Python is a dynamic language and hence it does not compile your code like other programming languages such as Java. Pylint is the tool that provides details about programming errors, syntax checks, import errors as per the recommended PEP-8 style.

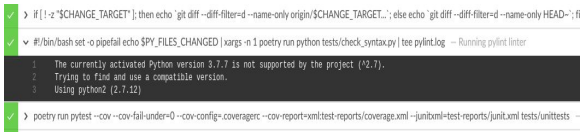



Figure 1: Output from pylint execution

**ii) Sonarqube:** Sonarqube is an apt tool for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities.

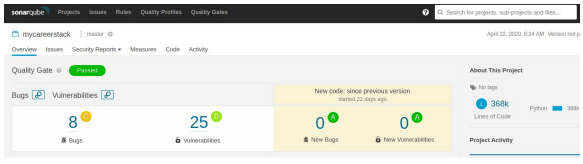



Figure 2: Sonarqube dashboard

## Unit tests

Unit testing involves breaking down the code segments into small bite size chunks or units of code or logic that can be quickly and easily tested. This makes sure that each unit developed works as expected individually. Here is the example of django-nose results of unit tests. We mandate the percentage of coverage to be more than 80% for the pipeline to proceed.

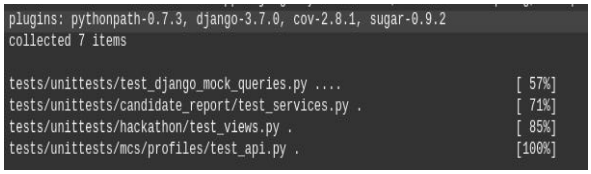



Figure 3: Output from unit test

## Integration tests

Integration testing is a type of testing in which components such as software and hardware are combined to confirm that they interact according to expectations and requirements. Certainly few things are not in the scope of Unit Testing and hence need to be addressed in the Integration Testing. Eg. Workflow from one component to other components. We are using django-nose for the same.

## Deployment

We have written a python script that builds the code from release and runs collectstatic to bump the versions of JavaScript files. The script finally deploys the application on uwsgi servers with nginx. Jenkins is the orchestrator to trigger this script and deploy the code.

## Automated Functional tests

We use pytest with the selenium driver to run our automated smoke and regression functional tests. The smoke tests are written to verify the exit criteria for user stories. If smoke tests fail, it simply means that the stories are missing exit criteria. These tests get triggered every time a code gets deployed to staging and production environments and shares the html results.

We have 200+ functional tests which we run parallely using pytest xdist and the execution time is reduced by around 65%. The distribution of tests in different nodes is written to scale for dynamic numbers of nodes available.


Figure 4: Pytest execution summary

## Security tests

Once your code quality is verified, and the application is deployed to a lower environment like Staging or QA, the developed code requires compliance to make sure there are no security vulnerabilities in the running application.

OWASP Zed Attack Proxy (ZAP) is one of the world's most popular web application security testing tools. It automatically finds security vulnerabilities in your web applications while you are developing and testing your applications. It can be used as a man-in-the-middle between browser and app server, as a standalone application, or as a daemon process without UI. We are currently running zap with default policy and are in the process of creating HackerEarth specific policies.

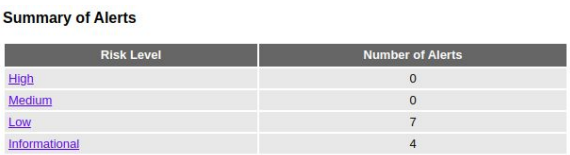



Figure 5: ZAP vulnerability report

**Performance tests**

Performance tests can be really useful to determine if the APIs developed are as per the benchmark. These tests give us the details about the time taken and responsiveness of our application. We are using locus for performance testing to swarm huge numbers of users and check if application responses are within expected time.

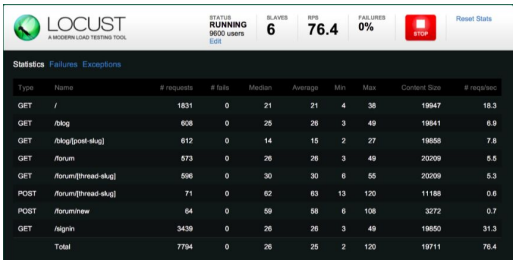



Figure 6: Locust dashboard

## OUTCOMES

Some of the significant yields post our implementation on Oct'19 are as follows:

- ✓ Quality of the software is directly linked to the number of the bugs in the development cycle and we have achieved close to 70% reduction in bugs (Q4'19 - **29**, Q1'20 - **18**)
- ✓ Faster and easy to deploy in just a touch of button with a total elapsed time of 45 mins without any manual intervention.
- ✓ Elapsed time to detect and correct production escapes is shorter with a faster rate of release.
- ✓ Mean time to resolution (MTTR) is shorter because of the smaller code changes and quicker fault isolation.
- ✓ Testability improves due to smaller, specific changes. These smaller changes allow more accurate positive and negative tests.
- ✓ Secure and high quality production code
- ✓ Automated pipelines help lesser tech savvy people to take part in the deployment process.

## CONCLUSION

Our pipeline is more efficient and accurate when the rules are defined in each of the stages and they tend to vary from one organization to another based on their requirements. Developers simply push code to a repository and this code will be integrated, deployed, tested again, merged with infrastructure, and also they go through security and quality reviews, which are ready to deploy with extremely high confidence. CI/CD improves code quality, and software updates are delivered faster making it the best ever process.

## AUTHOR BIOGRAPHY

**Payal Moondra** is a pioneer in engineering best practices and that includes driving automation initiatives using Rest APIs, Pytest to help increase Safety net focussed towards orchestration of deployment using CI/CD tools like Jenkins. Prior to HackerEarth, Payal has worked at startups like [24]7.ai, Sahi where she helped build automation frameworks on Java and JavaScript. Beyond professional life Payal is an active member of Toastmaster International and writes actively at Quora (Payal-Shah-26)

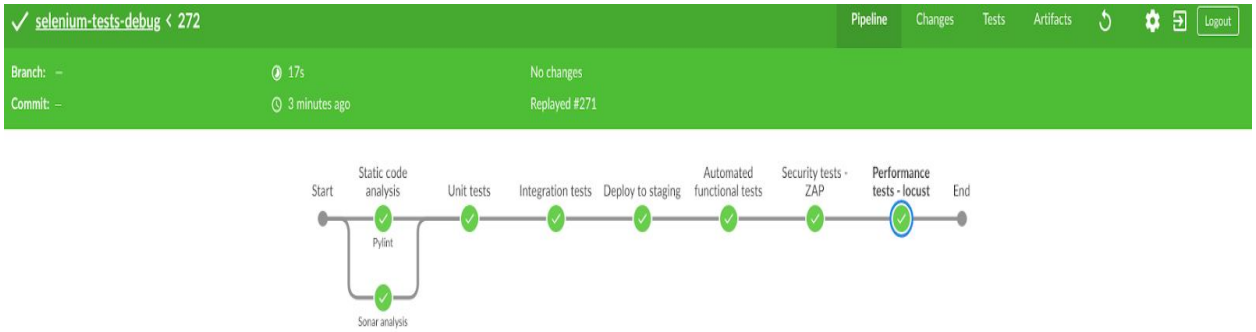
**Karthik Raju** is a specialist in test automation using UFT and Selenium and also holds framework knowledge. He is proficient in Java and Python and has worked on multiple cost saving initiatives in his team. He is winner of prestigious organization level awards and has been a top achiever in automation and hackathon events. He has won accolades for his technical skills across India and the United States.

## REFERENCES/BIBLIOGRAPHY


i. https://www.functionize.com/blog/why-test-automation-is-essential-for-ci-cd/
ii. https://nullsweep.com/dynamic-security-scanning-in-a-ci-zap-scanning-with-jenkins/
iii. https://services.google.com/fh/files/misc/state-of-devops-2019.pdf

## IMPLEMENTATION ARTIFACTS

## COMPLETE PIPELINE in JENKINS (Blue Ocean for UI)




## TOOLS AND PLUGINS

| Stage | Tool | Plugin | Version |
|---|---|---|---|
| Static code analysis | Pylint | - | 1.6.3 |
| | Sonarqube | - | 7.5 |
| | - | Sonar scanner jenkins plugin | 4.3 |
| Unit tests | django-nose | - | 1.4.5 |
| Integration tests | django-nose | - | 1.4.5 |
| Functional tests | Pytest | - | 5.3.4 |
| | Xdist | - | 1.5 |
| Deployment | Jenkins | - | 2.204.1 |
| Security tests | ZAP | Official OWASP ZAP | 1.1.0 |
| Performance tests | Locust | - | 0.14.5 |