Modernizing Continuous Integration

By: Ravi Lachmann . Tech Evangelist

© Harness Inc. 2021

Table of Contents

Summary

When talking to organizations, at first glance, Continuous Integration seems to be a solved problem. The demand for increasingly distributed applications has risen with the bloom of microservices, and development teams have the expectation that every commit should be a build. Continuous Integration has surprisingly become an unexpected bottleneck.

In this eBook, we will go through the pillars of Continuous Integration and how to modernize your Continuous Integration practices and platforms.

What Is Continuous Integration?

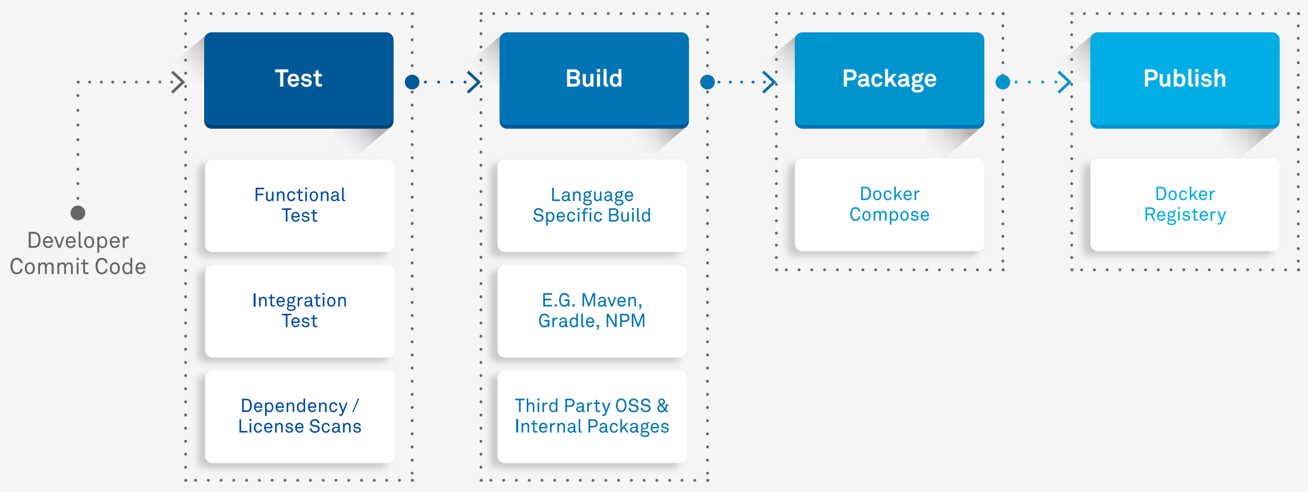
Simply put, Continuous Integration is build automation. Though, more than just the compiled source code can go into a build; the end product for Continuous Integration is a release candidate. There could be quality steps taken to produce the artifact, such as finding bugs and identifying their fix. Packaging, distribution, and configuration all go into a release candidate. For example, a JAVA application has a JAR, which is then packaged into a Docker Image, and has available all of the environmental configurations the image needs to run. A release candidate is the final form of an artifact to be deployed.

Why Use Continuous Integration?

Having an artifact that is ready to be deployed and/or continued to be vetted (for example, from development to quality assurance environment) is prudent in today's software engineering organizations. The main output of a software engineer's work is iterative by nature. Several artifacts can be created before a viable release candidate is made. The ability to build on-demand and start the integration and quality journey begins with a build that can happen multiple times per day. According to Paul Duvall, co-author of Continuous Integration, in a nutshell, CI will improve quality and reduce risk.

—

Typical Continuous Integration Process

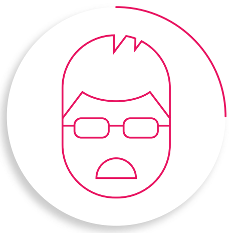


Benefits of Continuous Integration

Having an automated Continuous Integration approach frees teams from the burden of manual builds and also makes builds more repeatable, consistent, and available. Having the main work product of a software engineering team (the deployable unit) ready to be deployed regularly is beneficial to the entire software delivery life cycle and allows for consistent collaboration between engineers by avoiding common bottlenecks.

Repeatability

Externalizing the build instead of being only executed locally by a developer puts more eyes on the build steps. The Continuous Integration configuration starts to have less of an individual snowflake approach and can be an asset the broader team uses. Having a build executed by a system is repeatable and a march towards having consistency.



It worked on my machine yesterday...

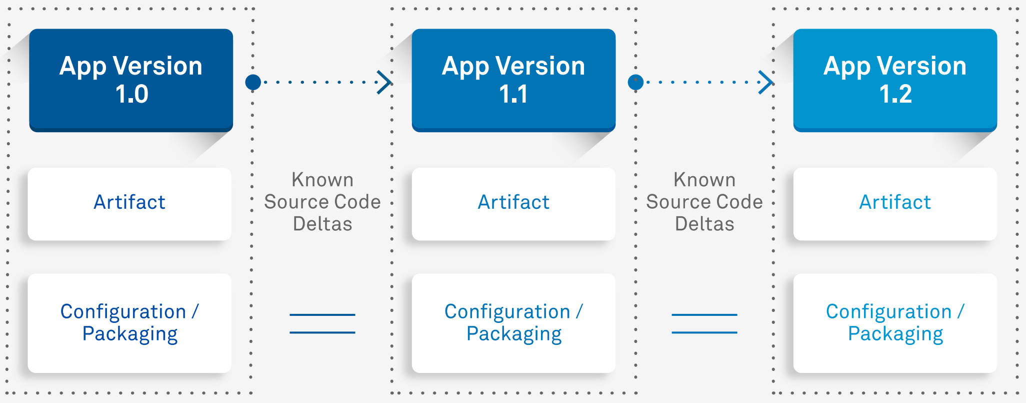
It works on my machine AND other machines today and tomorrow.

Consistency

The ability to build consistently is one of the major pillars of Continuous Integration. After having repeatable builds, efficiency and consistency start to step in as Continuous Integration practices become more mature on the team. Having a consistent build/artifact is key with maintaining dev-to-prod parity. An example of this would be keeping the environments similar or having the changes between them well-known. With consistency, there is also the ability to have builds more readily available.

—

Consistent Building & Packaging



Benefits of Continuous Integration (Cont...)

Availability

The ability to scale to match the demands of the concurrent builds needed by a team, and the ability to recreate a build, is the availability of the build. Modern containerized builds require more horsepower than just building the application binary. Distributed build systems allow for those builds to be more available. Since the builds are repeatable and consistent, a core tenet of modern software development is to be repeatable at any step of the process. Old builds and previous versions can be available by simply calling a recipe from the past. With the emphasis on having a build available at any time, challenges can arise supporting a wide swath of technology.

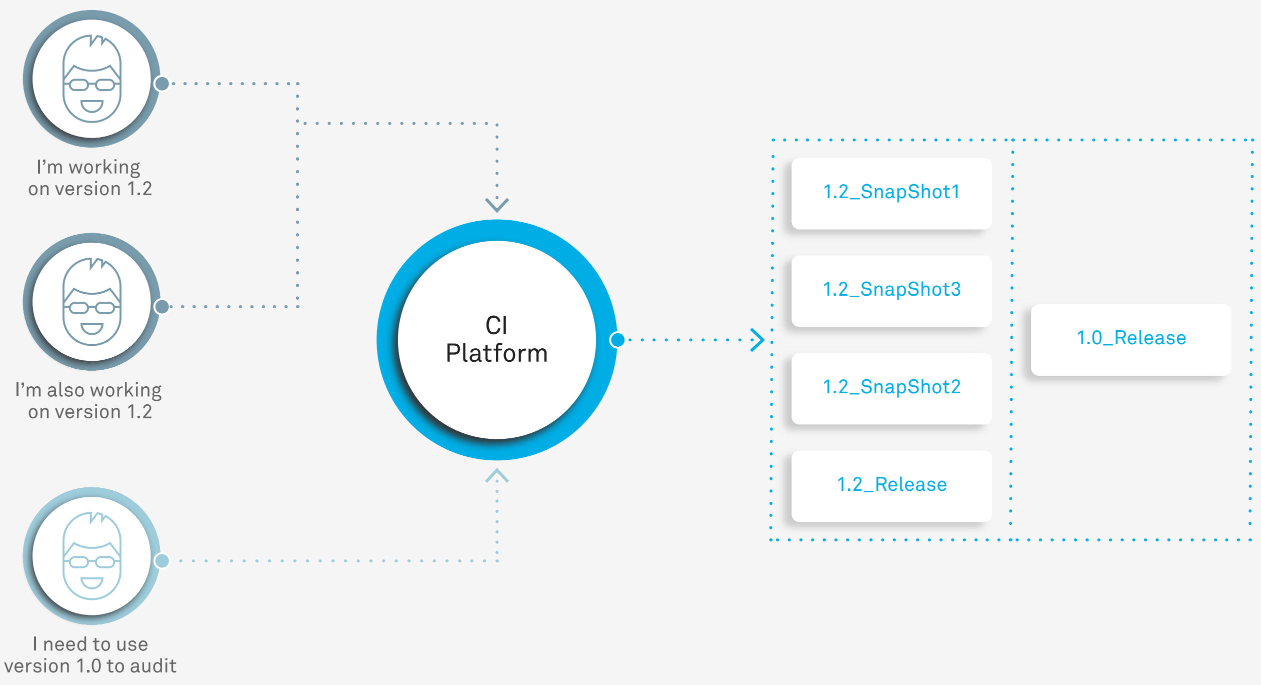
Continuous Integration Best Practices

As Continuous Integration continues to evolve, certain practices lend themselves to a more mature Continuous Integration approach. A mature Continuous Integration practice should allow for speed, agility, simplicity, and allowing for the dissemination of results in an automated fashion.

Keep the Automated Build Fast

Since builds will occur throughout the day, having a speedy and automated build is core to engineering efficiency. Not tying up an engineer's machine or local environment for a build by having a build externalized can allow an engineer to continue to make strides and adjustments as a build occurs.

Ability to Create a Build of Any Version at Any Time


I'm working on version 1.2
I'm also working on version 1.2
I need to use version 1.0 to audit
CI Platform
1.2_SnapShot1
1.2_SnapShot3
1.2_SnapShot2
1.2_Release
1.0_Release

Continuous Integration Best Practices (Cont...)

Keep the Automated Build Fast (cont...)

Simply put, the quicker the build, the quicker the feedback can be implemented—or a release candidate created to be deployed—by a Continuous Delivery solution.

Every Commit Should Be Built Automatically

For a software engineer, a commit—or merge, for that matter—in a shared repository signals moving forward in the software development life cycle. With a commit, you are committing that you are ready to start trying out what you developed. Core to Continuous Integration is to treat each commit as a potential release candidate and start building the artifact. This will allow for less lead time when a decision is made to deploy.

Small Pieces

In microservices and in Continuous Integration, smaller pieces can help reduce complexity. By having smaller and functionally independent pieces such as build, testing, packaging, and publishing, the identification of problems/bottlenecks becomes much easier. If there are changes to any one of the functional areas, they can be made and tweaked and the steps inside a Continuous Integration platform can be updated. With smaller pieces, if certain pieces need to run on other systems, finding the line in the sand to lift or migrate functionality is easier.

Be Transparent With Results

Feedback is crucial in the software development life cycle, and most likely, the first time changes are leaving an engineer's local environment is with a Continuous Integration process/practice. Disseminating build and test results across the teams in a clear, concise, and timely manner helps engineering teams adjust and march towards a successful release candidate. Initial builds are expected to run more than once as iteration occurs. Depending on the Continuous Integration platform, implementations can vary, especially around sharing results.

What Is the Difference Between Continuous Integration and Continuous Delivery?

Delivering software can be seen as continuous decision-making. Getting your ideas to production in a safe manner requires confidence-building exercises in the form of tests and approvals, and safe mechanisms to deploy, such as a canary deployment. Continuous Delivery is the ability to deliver changes to your users in an automated fashion. Continuous Delivery is interdisciplinary, bringing in automation practices around monitoring, verification, change management, and notifications/ChatOps. Without an artifact to deploy, there would be no deployment; Continuous Integration provides the artifact to deploy. However, Continuous Integration is not without its challenges.

Challenges of Continuous Integration

Because builds and release candidates follow advancements in development technology closely, such as new languages, packaging, and paradigms in testing the artifact, expanding the capabilities in Continuous Integration implementations can be challenging. With the introduction of containerization technology, the firepower and velocity required to build increased.

Challenges of Continuous Integration (Cont...)

Scaling Continuous Integration Platforms

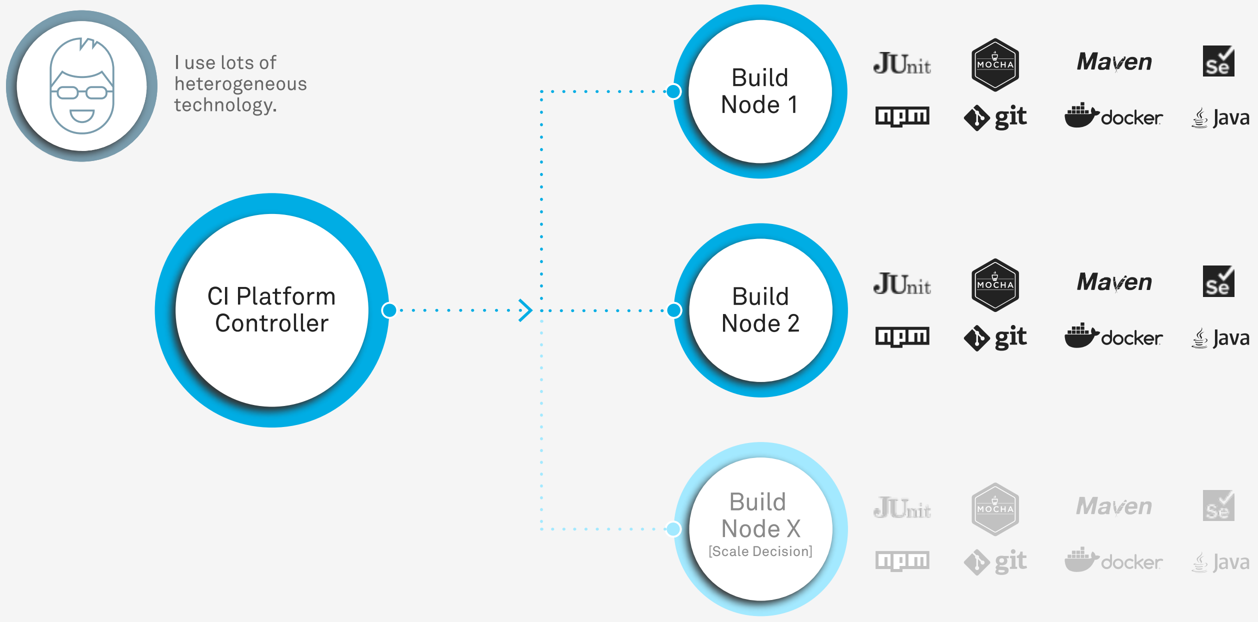
As the velocity of builds increases to match the mantra that "every commit should trigger a build," development teams could potentially be generating several builds per day per team member, if not more. The firepower required to produce a modern containerized build has increased over the years, versus traditional application packaging.

The infrastructure required to run a distributed Continuous Integration platform can be as complex as the applications they are building because of the heavy compute requirements. Take a look at how much of your local machine's resources are tied up during a local build and test cycle. Now, multiply that by the number of folks on a team or in an organization. Distributed build runners are one area that can be complex; managing when new build nodes are spun up and spun down can depend on the platform/end-user.

Keeping Up With the Technology Velocity

The adage "the only constant in technology is change" is true. New languages, platforms, and paradigms are to be expected as technology pushes forward. The ease of including new technologies in a heterogeneous build or accepting new testing paradigms can be difficult for more rigid/legacy Continuous Integration platforms that were designed for a small subset of technologies.

Build Runners/Nodes With All Needed Dependencies



Challenges of Continuous Integration (Cont...)

Homegrown/legacy Continuous Integration platforms can be very prone to rigidity, in terms of being designed for what was in the enterprise at the point in time when the platform was built. New technologies and paradigms require new dependencies for builds to occur or new testing methodologies to be implemented. Adding new dependencies should be as easy as the developer experiences on their local machine; e.g. simply declaring what is necessary and convention/ declarative-based tooling resolves the dependencies. With legacy or rigid approaches/platforms, dependency management required to maintain technical velocity is a significant burden.

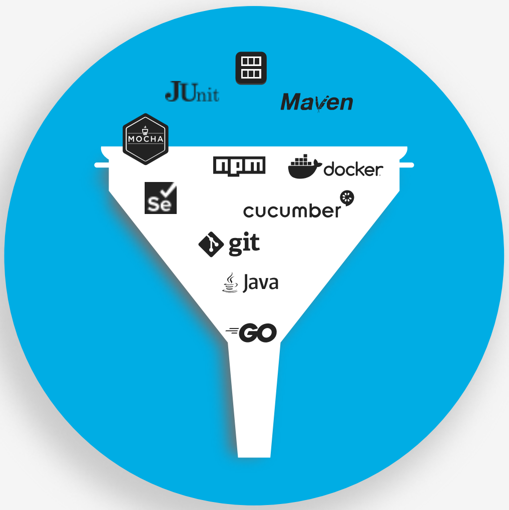
Overstretching CI Platforms Into CD

As some of the first systems that automate parts of the development pipeline, there would be a natural tendency to continue to build the automation that takes software all the way to production. Though organizations quickly realize that failing the build due to failing unit tests is different than handling multiple deployments and release strategies; a failed deployment can leave a system in a non-running state. This is why there should be a line in the sand between Continuous Integration (build) and Continuous Delivery (safe production deployments).

The rigor needed to create and test the infrastructure and application together, all while having a safe release strategy, such as a canary release, requires codifying tribal knowledge about applications to determine pass/failure scenarios. The burden of adding additional applications can be substantial and can go against best practices for Continuous Integration, such as keeping the build fast.

—

The Technology Funnel



Modernizing Continuous Integration

Modernizing Continuous Integration practices and platforms can take a four-pillar approach. Making strides in any of the pillars will put you on the path of modernizing your Continuous Integration platforms and practices.

Platform Infrastructure Modernization

Because of the speed, velocity, and concurrent nature that Continuous Integration solutions must be able to operate in, the platforms that run the external builds can mimic the applications they are supporting. With elastic infrastructure much more attainable today (for example, with autoscaling resources in the public cloud and/or by leveraging modern distributed platforms like Kubernetes), having a distributed and cloud-native infrastructure powering the CI platform is a necessity.

The location where the actual builds and packaging take place (e.g. build nodes/runners) do most of the heavy lifting and have a fairly elastic workload nature. Builds (e.g. a JAVA JAR build) and packaging (e.g. a Docker Image Compose) are a compute-heavy task. Once the build and packaging are complete, the runners can sit idle. This shows the importance of having an ephemeral build node. The build node spins up during the task then spins down/is destroyed after the build task is done, so as to not drain resources.

Even if the applications that are being built themselves are not headed to a Kubernetes endpoint, running your Continuous Integration solution on Kubernetes can provide learnings for the organization on how a distributed application runs on Kubernetes. Modern Continuous Integration platforms are designed to have ephemeral nodes/runners being deployed to elastic/Kubernetes-based infrastructure. Running on modern infrastructure also has benefits in engineering efficiency.

Engineering Efficiency Strategy

A core tenet of engineering efficiency is meeting your internal customers where they are. For software engineers, this is being as close to their tools and projects as possible. Like many modern pieces of application infrastructure, shifting left to the developer means being included in the project structure in source code management (SCM).

For local builds, checking in to source control language-specific build files, such as Maven, Gradle, or NPM configurations, has been the convention for some time. Though with additional packaging, confidence, and build steps (for example, more than one language or artifact the distribution), Continuous Integration platform steps are now being included in SCM-managed projects. Modern Continuous Integration platforms support declarative instructions where goals are defined and the CI solution will work to create the desired declarative state (e.g. the output artifacts).

A common disconnect in Continuous Integration platforms is dependency management. Over the past decade, for software engineers, this problem has been solved with dependency/package/build tools such as Maven, Gradle, and NPM. Simply define implicitly or explicitly what you need, and the dependencies will be resolved. Continuous Integration tools suffer from a disconnect since we are leveraging several tools that potentially don’t have a common syntax.

Modernizing Continuous Integration (Cont...)

Engineering Efficiency Strategy (Cont...)

Dependency management between the build nodes and runners is a common pain point. For example, certain nodes have certain dependencies, and certain nodes don't. Modern solutions take a container-based approach (e.g. Docker-based) with dependency management in the ephemeral build node/runner container. Declare what you need, and similar to a Docker build/compose, it will be executed, giving the node container everything it needs.

Test Optimization

Typically, an organization's first forays in running automated tests in a repeatable and consistent fashion end up in their Continuous Integration pipelines. Usually, this is an easy lift; the same code/test coverage that a developer is subject to in their local build makes its way into the build pipeline since those steps should have been executed before the commit.

Though as the initial confidence of getting tests into the CI pipeline expands, more tests and sometimes inappropriate test coverage is introduced due to the ease of integration with the pipeline. An even harder problem to identify and rectify are flaky tests. A flaky test is a test that both passes and fails periodically without any code changes. A twofold problem of increasing execution time and lack of confidence with flakiness requires optimization to avoid. A modern Continuous Integration solution should be able to visualize order, timings, and overall execution to help identify and eventually rectify excessive coverage and flakiness.

Rapid Pipeline Development

Software is an exercise in iteration. The lower the barrier of entry for iteration to occur, gains in engineering efficiency and agility are achieved. Local builds happen dozens of times before reaching a committable stage; moving forward to a dev-integration environment. Having a local environment is key. Oddly, Continuous Integration pipelines are designed to run externally from a local machine; that is the entire point.

With a chicken-or-the-egg problem, a CI pipeline needs to be developed before being run/accepted. Usually, CI pipelines are developed remotely since the CI systems are remote to a user's machine. The ability to have localized CI pipeline development allows for the same iteration velocity that software engineers have been achieving for a while. Also, locally-run CI pipelines allow the internal customers (the software engineers) to run and debug pipelines before making a commit that would trigger a build, therefore building confidence before a build.

Modern Continuous Integration Architecture

Modern Continuous Integration architecture supports iteration and scale while optimizing and building confidence, all while enabling proper feedback loops that are in place for action and automation to take place.

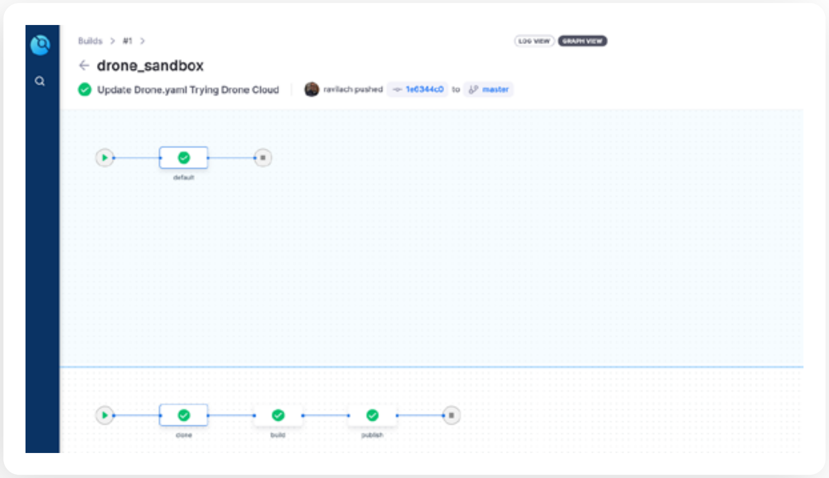
With the following model/architecture, Harness Continuous Integration can easily be deployed, and it supports modern Continuous Integration approaches.

Modern Continuous Integration Architecture (Cont...)

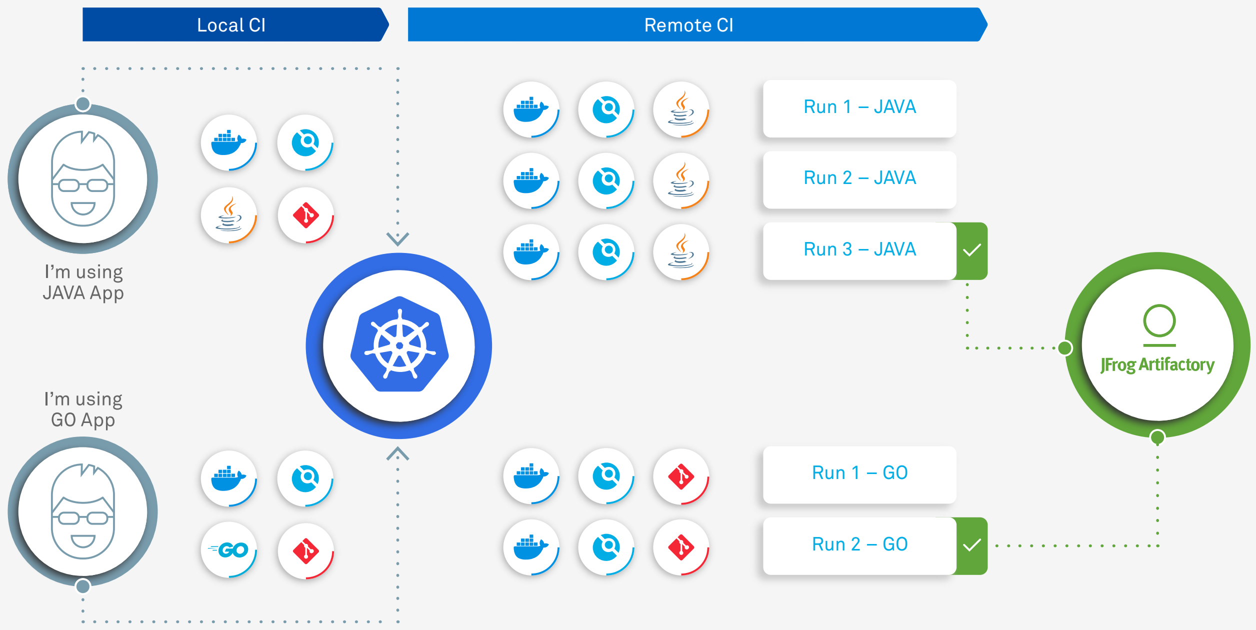
Harness Continuous Integration

Harness Continuous Integration, both Enterprise and Open-Source (based on Drone), have modern user interfaces and are built to meet the scaling requirements of cloud-native workloads.

Graphical breakdown of pipeline and decisions in the pipeline.

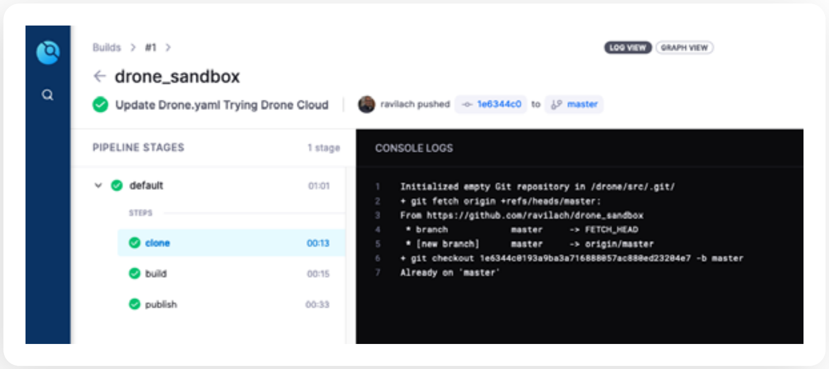


Modern CI Execution Diagram



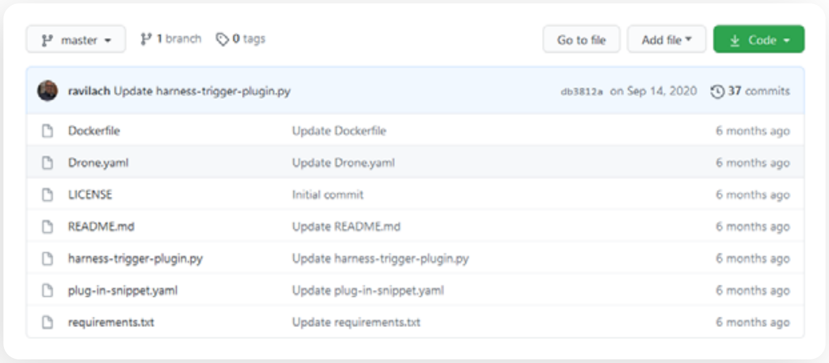
Modern Continuous Integration Architecture (Cont...)

Sequential breakdown of progression of the pipeline.



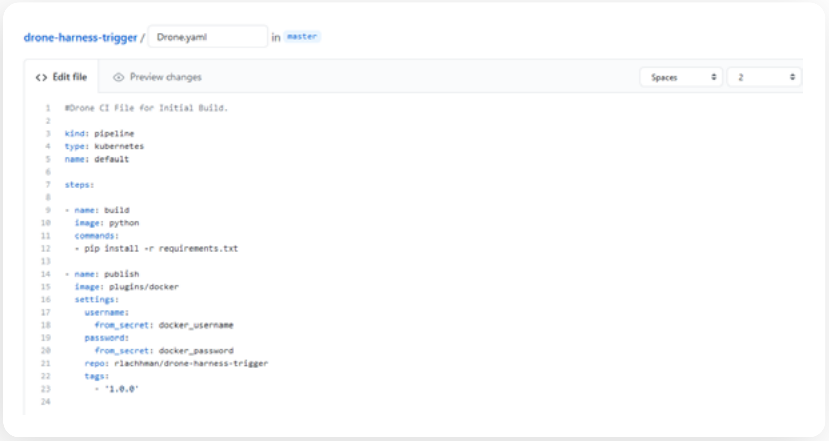
```
Initialized empty Git repository in /drone/src/.git/
+ git fetch origin +refs/heads/master:
From https://github.com/ravilach/drone_sandbox
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> origin/master
+ git checkout 1e6344c0193a9ba3a716888057ac880ed23204e7 -b master
Already on 'master'
```

Changes live in the project structure and have the ability to be triggered by the SCM.



Dockerfile	Update Dockerfile	6 months ago
Drone.yaml	Update Drone.yaml	6 months ago
LICENSE	Initial commit	6 months ago
README.md	Update README.md	6 months ago
harness-trigger-plugin.py	Update harness-trigger-plugin.py	6 months ago
plug-in-snippet.yaml	Update plug-in-snippet.yaml	6 months ago
requirements.txt	Update requirements.txt	6 months ago

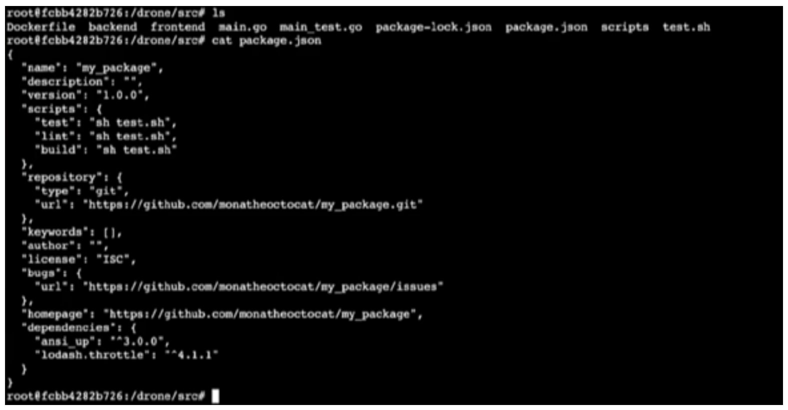
Simple and declarative configuration which lives in the SCM.



```
#Drone CI File for Initial Build.

kind: pipeline
type: kubernetes
name: default

steps:

- name: build
  image: python
  commands:
  - pip install -r requirements.txt

- name: publish
  image: plugins/docker
  settings:
    username:
      from_secret: docker_username
    password:
      from_secret: docker_password
    repo: rlachhman/drone-harness-trigger
    tags:
      - '1.0.0'

```

Modern Continuous Integration Architecture (Cont...)

The ability to locally or remotely debug a pipeline.



```
root@fcbb4282b726:/drone/src# ls
Dockerfile  backend  frontend  main.go  main_test.go  package-lock.json  package.json  scripts  test.sh
root@fcbb4282b726:/drone/src# cat package.json
{
  "name": "my_package",
  "description": "",
  "version": "1.0.0",
  "scripts": {
    "test": "sh test.sh",
    "lint": "sh test.sh",
    "build": "sh test.sh"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/monatheoctocat/my_package.git"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/monatheoctocat/my_package/issues"
  },
  "homepage": "https://github.com/monatheoctocat/my_package",
  "dependencies": {
    "ansi_up": "^3.0.0",
    "lodash.throttle": "^4.1.1"
  }
}
root@fcbb4282b726:/drone/src# █
```

In Conclusion

Continuous Integration might seem like a solved problem for many organizations, but as with any technology, there is always room for improvement and modernization. With modern development processes allowing for more rapid development, the platforms that support the agility and iteration that organizations require are evolving. Legacy approaches are seen as brittle and rigid, and incorporating modern practices and approaches into your Continuous Integration platform will allow for future growth and agility in a lasting solution.

Author Appendix

Written By:

Ravi Lachhman, Evangelist at Harness
Ravi Lachhman is an evangelist at Harness. Prior to Harness, Ravi was an evangelist at AppDynamics. Ravi has held various sales and engineering roles at Mesosphere, Red Hat, and IBM helping commercial and federal clients build the next generation of distributed systems. Ravi enjoys traveling the world with his stomach and is obsessed with Korean BBQ.

Reviewed By :

Bryan Feuling, Solutions Architect at Harness
Bryan began his tech career as a Help Desk Technician for a Fortune 200 company. His experience includes Database Administration, Application Development, Automation Engineering, and more. He has worked with hundreds of companies to help them avoid the same issues, pains, and engineer burnout that he saw and still sees in the industry.

© Harness Inc. 2021