CICD - Working Title

DeDominic, Anthony
Eastern Connecticut State University
Willimantic, USA
dedominica@my.easternct.edu

Abstract

To be written. to see latest revision, please see:

https://github.com/adedomin/cicd-in-dist-env.git

1. Introduction

Content goes here. Example use of cite-proc [1]

2. Background

CICD is the process of accelerating the building, testing and installation of applications to end servers . In a world where web technology moves fast and new features are ideal, it's critical to go to market fast. To do this there are two major architectures that are used.

CI, continuous integration is concerned with the development process. In the realm of continuous integration there is usually a source code repository. The source code repository is usually technologies like git, svn, cvs or other version and source controlling software. The idea of the source code repository is to allow for efficient ways to handle code changes and to manage multiple developers working on the same code.

Lastly, there is a server, running software similar to Jenkins CI or Travis CI which can potentially handle many tasks. At a high level, it is usually used to run a build or test step in, say, a build.gradle, project.json, Makefile or various other build tools. To enhance tools like Jenkins, there are binary and other code quality scans like SonarQube and others which attempt to find issues outside of the space of unit and mock tests. Other such triggers, like diff checking, can be used to trigger code reviews for massive revisions of source code.

Generally, the final stage of Continuous Integration is to notify the developers, or other interested parties, the results of these various tests and builds. In a full CICD build pipeline, generally there is the process of uploading so called artifacts to a artifact or binary repository. These binary repositories are more in the realm of continuous delivery.

Continuous delivery also has it's share of tools and structure. One of the first requirements of continuous delivery is having some form of binary repository. For java based projects, there are things like nexus. Other languages, like python, have their own packaging sites; python for instance has PyPI.

In order to deploy these binaries, many tools make use of remote shell protocols. Remote shell protocols give tools access to execute the needed steps to get software deployed. Generally these tools run through scripts which make the machine require the needed binaries and create the needed configurations. The remote shells are no different than conventional user-controlled shells.

There are various tools and servers that do this; a few examples are Ansible, uDeploy and Puppet. Ansible uses remote shell protocols, such as secure shell (ssh), to execute a set of instructions written in yaml, generally with the intent of installing some type of software. uDeploy is similar in how it causes change on target machines, using secure shell; however uDeploy generally requires the target machines to be running some kind of daemon.

Many POSIX-like systems make use of package management to deliver software. Packages, like .deb's and .rpm's, are a collection of installation shell scripts, including pre-installation and post-installation, and a binary or source code. Package managers, as their name implies, also manages these installed binaries, scripts and configurations. Part of their management requires tracking file locations. This allows for rolling back, upgrading or removal of installed files.

Deployment strategy can be different depending on the available infrastructure. In an environment with fixed inventory, it might not be ideal to do things like 'make install' which are hard to reverse. However in an environment where machines can be spun up at whim, or even automatically, it wouldn't matter as much; in such cases one would simply destroy and create a new machine for every deployment. In a fixed architecture, it

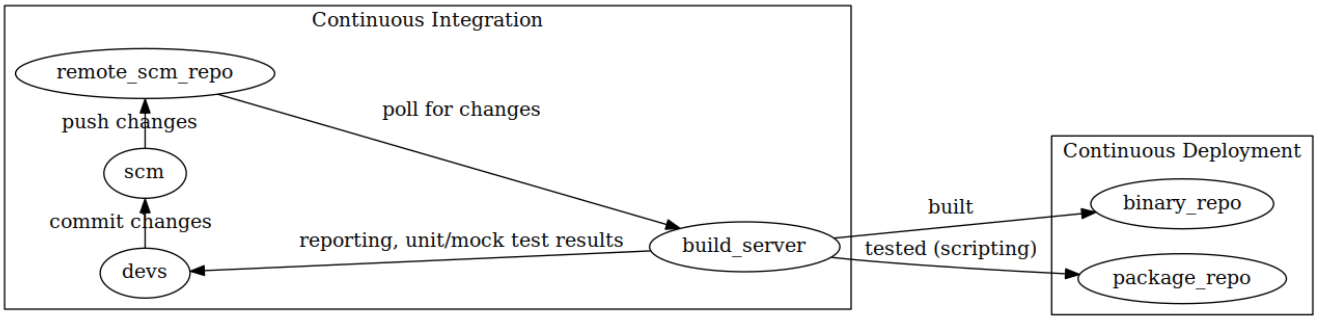

Figure 1: An example Continuous Integration Workflow

might be ideal to stick to package management solutions for delivering binaries. It might not matter if it is dynamic as the cleanliness of the system.

3. Ansible and other Domain Specific Languages

Ansible is one of the newer configuration management tools. One of it's advantages over it's predecessors is that it is agent-less; agent-less as in, it creates self-contained python payloads that is executes on the remote machine. This means less setup as other tools require a running service to manage and trigger jobs. However, the one thing that makes them the same is how they all kind of work.

Ansible, like many of it's competitors: puppet, chef, uDeploy, etc; consume domain specific languages. The domain specific language is then parsed and compiled (or interpreted) into some actionable set of instructions. To make them more extensible, many of these tools allow for creating what are called modules; These modules are written in a much more general programming language and are thus, more expressive, but theoretically harder to write for non-programmers.

Domain specific languages differentiate themselves from general purpose languages in the sense that their scope is significantly limited to a very specific problem. The goal of this is to make it simpler to solve problems in a particular domain by doing less. Generally, domain specific languages make use of simple grammars, like yaml, json, etc; to write in. These languages have mature parsers in most general purpose languages like python, ruby and perl. Thus, instead of writing a whole new language, generally one can fallback on these simpler languages and then generate the code using the structured output of the parsers.

Ansible for instance uses yaml syntax to describe "playbooks;" basically a set of instructions to execute on machines. The yaml is parsed into a simple object that is a collection of key value pairs, where the values can be numbers, strings, boolean, objects or lastly, arrays of all of the other types. From there, the ansible tool creates a python script by using the data contained in the parsed result. Calls to modules and other such things in the parsed structure are replaced with literal python source. Effectively, it's just a translation to library calls.

3.1. In CI

Travis CI makes use of a domain specific language to configure* one time build and testing environments for programs. Using a yaml source file, the developer can specify the needed build, and environment settings to define tests to be executed and ran. This file is then translated using ruby's extensive metaprogramming facilities into actionable code on Travis-CI's systems to build virtual machines to carry out the instructions laid out.

Jenkins has it's own descriptive language which allows it to do something similar to travis-ci. However many claim the richness of the language, especially when working with plugins in Jenkins, can make it unwieldy and challenging to use. It ultimately shows that a domain specific language should be very straightforward for it to be useful.

4. Downsides to Ansible, et al.

One of the biggest issues with these deployment tools, like ansible, is their irreversibility. Changes made to remote machines are not easily tracked and are thus harder to undo.

[1] more akin to a uDeploy setup, a master server orchestrates actions to be take against numerous machines; generally the machines fetch the binaries they need to install from an artifact repository. Master can be triggered to action by the CI system as well.

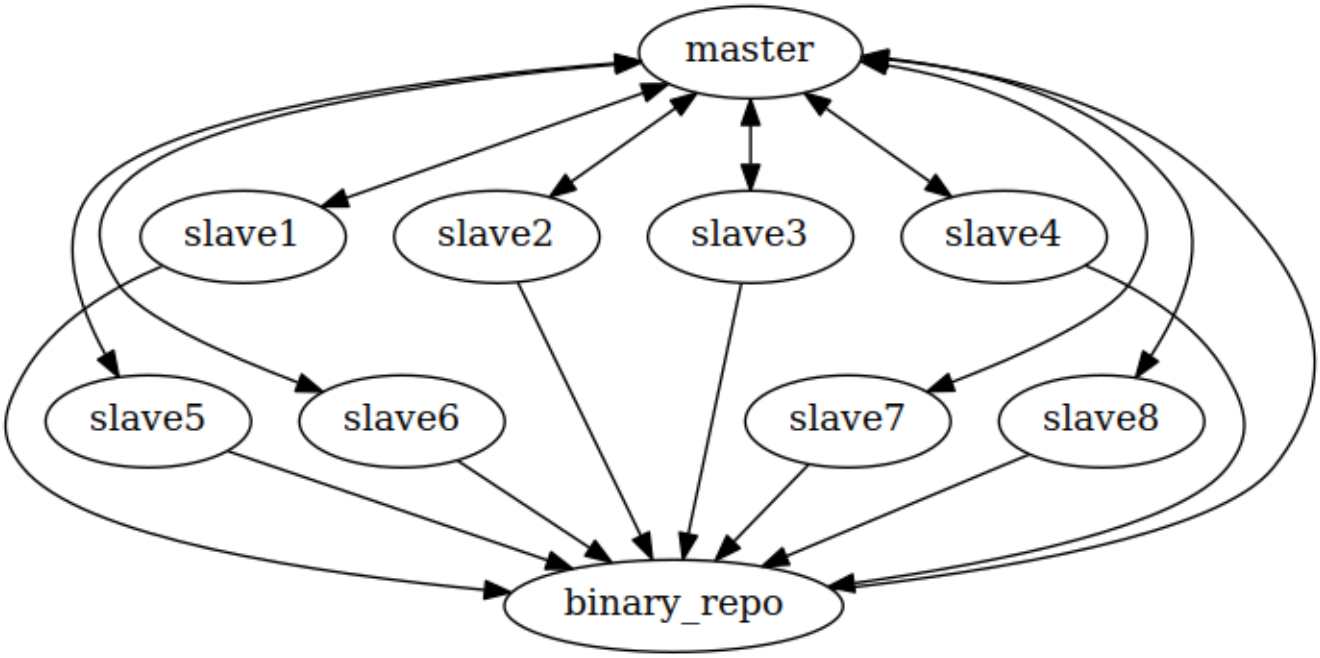

Figure 2: Example Continuous Deployment setup[1]

Another downside to domain specific languages as a whole is balancing the right amount of features.

5. Limitation of Current Domain Specific Languages and Tools

Below will be the explanation of a new DSL that I will develop that will attempt to solve shortfalls in other tools. Ideally, to also merge them into a universal testing, packaging and deployment framework. The goals of the language are to solve the following shortfalls:

5.1. Irreversibility

As Stated earlier, many deployment tools make large numbeer of untraced changes to the filesystem. Such changes as creating directories, inserting configs, binaries, scripts and other such files. Other changes, like new users, added and enabled servicesand other such changes to configurations are also untracked.

What is proposed is a simple key-value like database which can contain package name as keys and a value which is a document of changes -(more detail)-.

5.1.1. Detractions

The problem is, many CICD, or "DevOps", manuals expect you to do A-B deployments to prevent reversibility issue -(link to red hat customer portal)-. This way, when issues arises, the DevOps engineer can simply destroy the new machine and go back to the old one. Vise versa if the new machine is successful, the old one is destroyed. To take it to an extreme, using docker and docker container OSes like CoreOS and Red Hat Atomic Linux, The whole operating system is to be considered immutable and any updates should destroy the previous state of the machine.

5.2. Build and Testing Tool Dependencies

One of the biggest problems in the CI space is dealing with the demands of developers and the tools they need. Jenkins for instance is very difficult to work with when developers try to use tools that aren't available on the jenkins master, or worse, the slave. As a result, self hosted CI tools like Jenkins depend on a very rich set of plugins that try to solve these "dependency hell" problems.

Certain build tools like Apache Maven and Gradle use the concept of "wrappers" which are usually shell scripts that "resolve" the build tool binary of the specified version. However most depend on a binary .jar file to accomplish this.

5.2.1. Provided Services

For testing, it may be ideal to have services such as databases. Currently, it's the same problem with build tools; either the server is provided, externally or locally. It would be ideal to handle this problem in the language as well.

5.3. Options in Deployments

Currently, tools depend on remote shell protocols, as described earlier; however, shell protocol security can make it costly to open connections and to send data to them. For security reasons, concepts like asynchronous callbacks, can be difficult to achieve via shell protocols; unless of course the server can ssh into the deployment server (I have never seen this in practice).

Ideally, for updating, there should be optional, web centric way of pulling changes. Currently ansible is deployed manually, over shell. After the initial deployment, there could be a script which would allow for "pulling" updates over the network. To make ansible more as an automated service, projects like loopabull can add automation to the whole process.

5.4. "Artifacting"

The concept of artifacting is separating out the various components of an application into these versioned components. Currently only uDeploy, as far as I know, offers such a construct.

The advantage of this is deployments are broken into logical pieces. The Other advantage is the pieces are versioned as well; This gives a user the ability to fine tune the product to find the best composition of versioned artifacts.

5.5. Order of execution

Tools like ansible and puppet depend on tasks executing in a particular order. To attempt to add ordering, they add concepts like listeners and handlers. However, it can be confusing at time and add unneeded complexity for simple tasks.

Ideally one would be able to use monadic constructs to add ordering, but to functionally encapsulate the side effect laden issues of deployment. This could also help solve the irreversibly problem.

DSL should feature:

- Multi-paradigm for describing deployments
  - graphs (Metamodeling - or Model driven)
    - Ideally with a drawing tool.
    - Alternatively, users could make graphs using the DOT language, or another simplier graph language.
  - sequential steps (yaml | toml | json | etc)
  - similar to ansible, puppet and many others.
- Dependency management for playbooks & components
  - think npm, pip, etc.
  - requires playbooks and components have a package file that describes what they provide.
  - TravisCI leverages weaknesses in this problem to push it's nonfree software as a service platform.
- Integration of all the key steps in a integration and deployment pipeline
  - Integration (testing) (see figure* 3, shows domain problem at a high level)
  - Packaging (artifacting)
  - leverage language specific packaging?
  - Deployment
  - symlink management (like GNU Stow)
  - Configuration Management (see figure* 4, shows proposed to manage configurations)
- "Agentless" service
  - Utilize builtins only
  - standard POSIX utilities
  - Scripting languages
  - common languages like python 2
  - bash
  - no daemons/services
  - other than sshd or other remote shell service
  - pretty much ubiquitous

6. Domain Specific Languages (DSL)

- DSL
  - Limited scope,
  - easy to solve singular problem
  - Flex/Bison
  - shell utils: sed, awk, etc.
  - Composed into general languages
  - IR -> python, javascript, shell
  - Abstracted library calls
  - Similar to PHP's origin
- Challenges
  - scope creep
  - defining a encompassing scope for the problem domain
  - Adding logic to DSL

[2] graph that shows wha the Domain Specific language must solve in CI space.

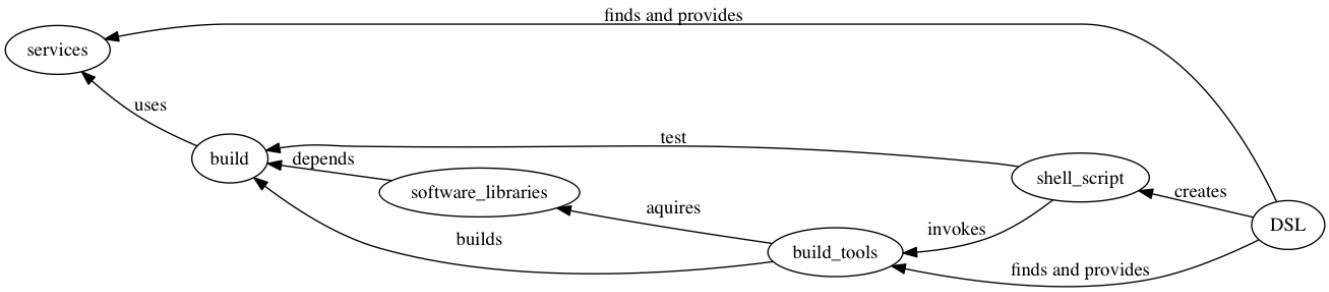

Figure 3: CI language domain[2]

6.1. Continuous Integration Domain

Key domain concepts

- shell driven interactions
- fork() and exec() of build tools.
- managing environment variables
- provisioning services, such as databases for testing purposes

Most of the problems in this domain boil down to shell script execution and interactions. Secondary problem is resolving dependencies for services, like databases; and build tools like gradle.

6.2. Configuration Management Domain

The final domain of the CICD process is configuration and maintenance. This portion is more of a optional "agent-less" manager. In this domain, the key problem is generating configuration files, and at a later time, able to regenerate them using newer information. Currently to solve this problem, configuration management tools (and by extension, continuous deployment tools) offer the ability to take a structured document and apply the values therein to a general template.

Citations

[1] M. Fenner, "One-click science marketing," *Nature Materials*, vol. 11, no. 4, pp. 261–263, 2012.

[3]The idea behind this, is instead of pushing plaintext configurations and basic templating, users can develope parsers for various software configs or derive them from the community. The parsers would allow for dynamic configuration to occur in a clean way. It would also allow for more advanced querying.

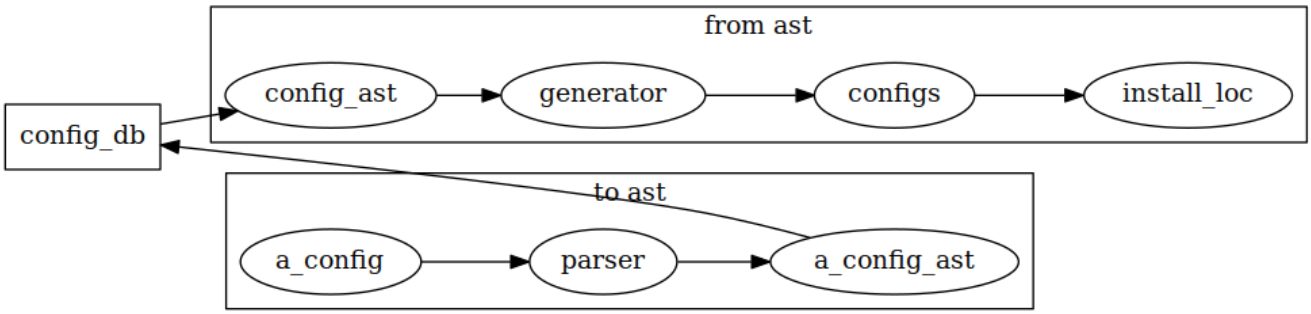

Figure 4: Example Workflow Proposal[3]