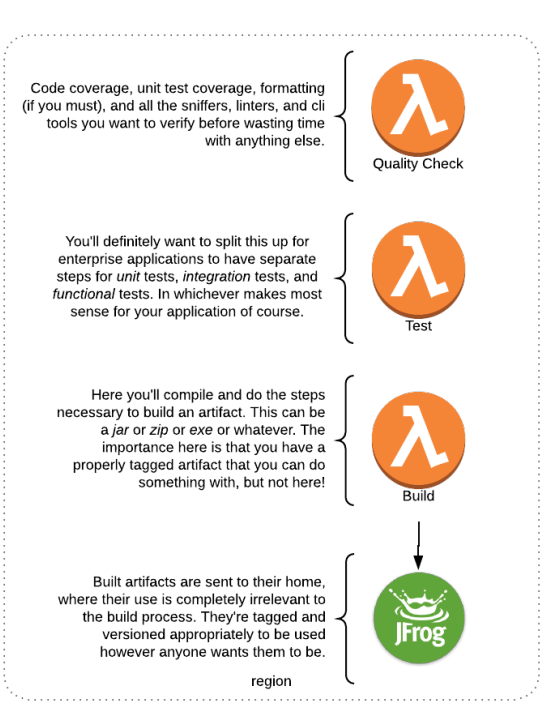
This is the first of four eventual posts that will articulate the approach my teams take to code pipelines and what I have identified as the four factors of an effective software development pipeline.

Factor #1: Separate Build, Deploy & Provision

BUILD

This sounds obvious, but how many of the engineers reading this have Jenkins “builds” that both build and deploy code in some way? How many times does a deployment to staging (or anywhere) fail because of an issue building code?



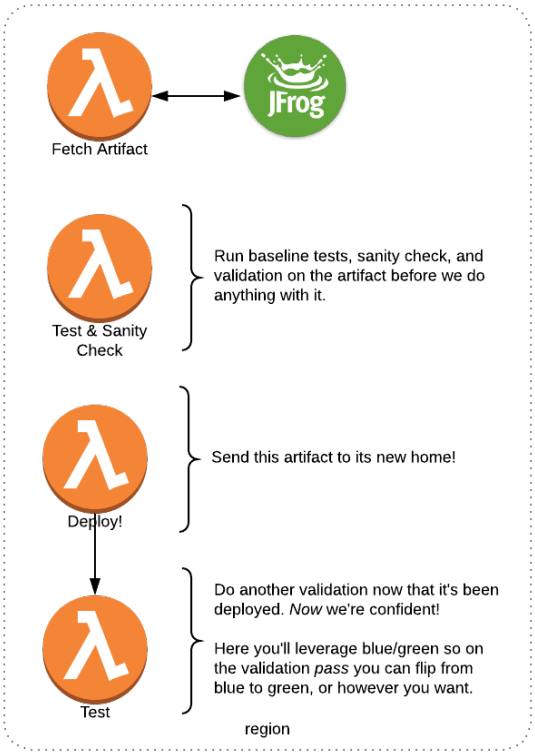
Worse yet, not all tech-stacks even require a true build; PHP and Python applications for example; and to a lesser extent the universe of Javascript. None of those applications need a compiled artifact to deploy, and the deployment generally speaking is just the aggregation of all code at a specific point in time; something `git` has made *extremely* simple.

To make the most appropriate use of your *builds* you should leverage an artifact repository. This could be as simple as a S3 bucket in AWS, or as complex as a JFrog artifactory. At this point, your build is either a compiled artifact, or a compression of all code as a single tagged,

versioned file.

For example; `{appname}-{branch}-{timestamp}.jar` for a Java application, or `{appname}-{branch}-{timestamp}.zip` for a PHP, Python, Javascript etc application. Where the zip file contains all the *.ext files for your application.

Now, over a lengthy period of time and depending on the size of your team and application, your storage of these artifacts will necessitate some discipline around cleaning those files up over time like you would any other storage.



DEPLOY

At the end of the day deployment is the simplest part of software. Or at least it should be,

right? The idea here is that we take a thing and put it in a place, and then tell our place to run our thing. What becomes essential in this part of the process is keeping your deployment very straight forward.

The best analogy I use frequently is that of a car manufacturer.

Assume for a moment that all homes in this analogy are identical, and that our deployment is the transportation of our vehicle off the assembly lot to the purchasers home. Our deployment in this analogy is simply the transportation of a specific VIN to a specific address. The truck transporting the vehicle should only need to be concerned with the specific vehicle (VIN) to deliver, and the precise address to deliver it to. This is precisely how we want our software deployment to function.

At that point, your deployment process is merely giving a function a pointer to a file, and an address to send it to in the simplest case. Additional steps of uncompressing and enabling daemons or scripts will of course be dependent on your use case, but the core function of the process is the same.

PROVISION

Provisioning of servers is an entire industry in the software business. I have no delusions as to the complexity and nuance of this art form, except to *insist* that the provisioning of servers for *any* environment be **completely** isolated from your build and deployment except in the sense you deploy using orchestrated containers.

For example, with Docker orchestration using Amazon ECS or Kubernetes, we cannot deny the inevitability of code requiring specifics of the server to execute properly, but they are and should still remain separate builds, and the final artifact is simply the combination therein.

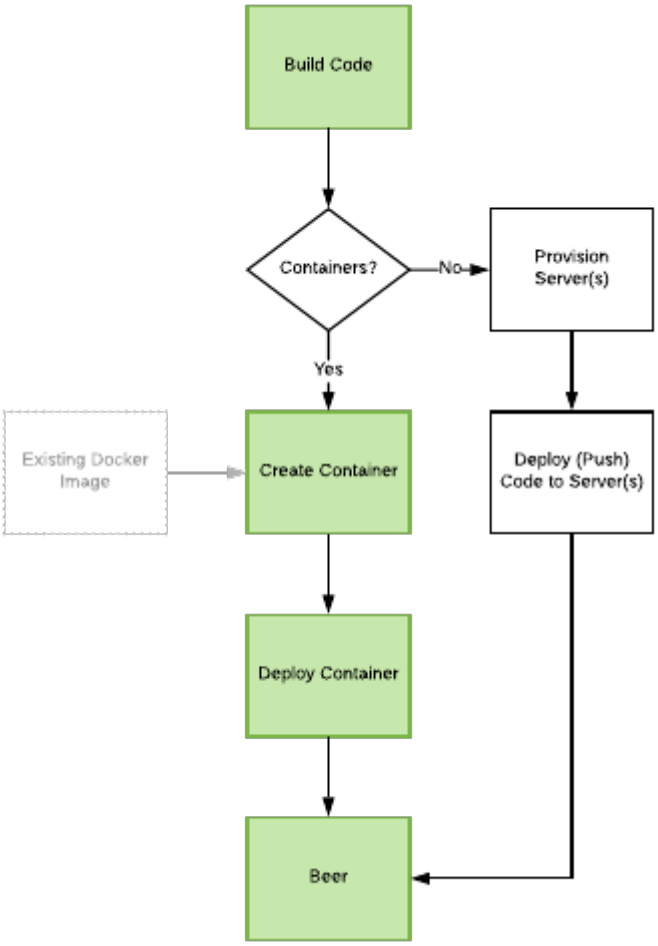
The goal is to view your server as part of the artifact you are deploying, and a Dockerfile should simply reference the appropriate image, and your deployment can reference that

accordingly. If you're not using containers, it's still equally important to provision separately from your build/deploy with a tool like Ansible or Chef to standardize your environments.

Using this same example, if you have a PHP application, your Dockerfile could look something like the following:

```
FROM alpine:3.6
LABEL Maintainer="Jake Litwicki <jake.litwicki@gmail.com>"
      Description="LEMP w/ Nginx 1.12 & PHP-FPM 7.1 based on Alpine."

# Install packages
RUN apk --no-cache add php7 php7-fpm nginx supervisor curl

# Configure nginx
COPY docker/nginx.conf /etc/nginx/nginx.conf

# Configure PHP-FPM
COPY docker/fpm-pool.conf /etc/php7/php-fpm.d/litwicki.conf
COPY docker/php.ini /etc/php7/conf.d/litwicki-php.ini

# Configure supervisord
COPY docker/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

EXPOSE 80 443
CMD ["/usr/bin/supervisord", "-c",
"/etc/supervisor/conf.d/supervisord.conf"]
```

This Docker image is then built and deployed separately to something like DockerHub, or Amazon Container Registry, tagged and versioned, and can then be referenced as `litwicki-app:{version}` in future deployments.



BUILD, Deploy & PROVISION, Oh My!

Naturally software is complicated. I am attempting here to simplify and articulate a simplification of a specific use case where you code *can* exist separately. This is an important decision to make as early on as possible so you can support this kind of abstraction and simplification.

Straight to the point, the primary tenet of this factor is simplified as the following:

1. Build your code by itself. Whether that means literally compiling, or running pre-compiling, sniffers, linters, etc, do it by itself, and when everything passes quality checks compress/minify and zip your artifact so you know what it is at quick glance.
2. If you can, use containers. Create your containers separately from your code build, and make as simple a Dockerfile as possible that pulls an existing image you've built separately with a similar isolated process.
   For example:

```
FROM litwicki-
app:{version|latest}

RUN mkdir /app
# unzip, uncompress or
manipulate your
# artifact for its final
resting place
COPY . /app
```

3. Deploy your code as a container.

Next: Skip The Gatekeeper

The next factor is to Skip The Gatekeeper