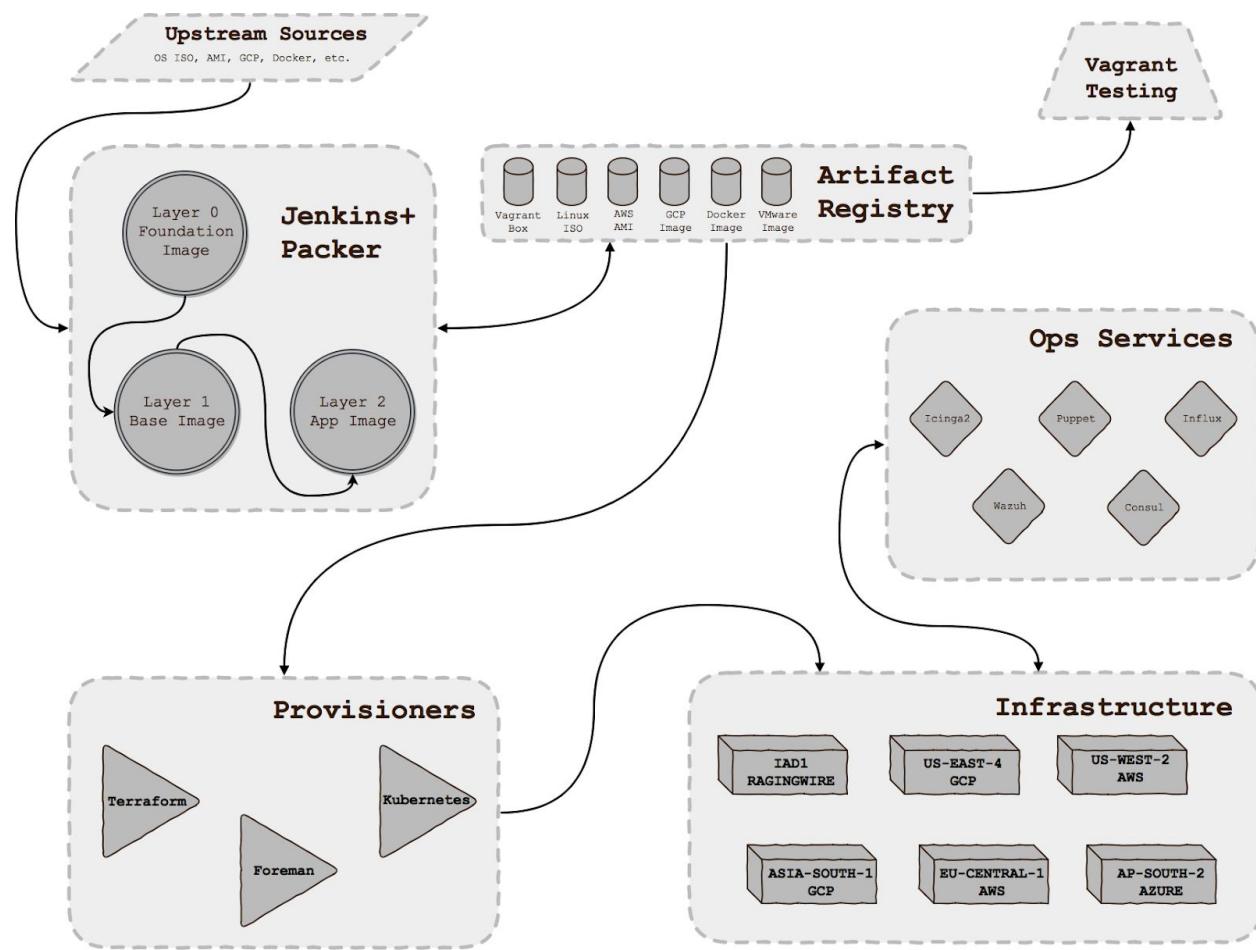


# Application Infrastructure Pipeline

## Overview

This document outlines the Application Infrastructure Pipeline from downloading from upstream sources and deploying applications all the way to a provisioning them on Bare Metal node, VMWare, GCP, AWS and even Docker containers. There can certainly be other permutations and combinations but the general workflow is as follows.



## Who uses this workflow?

This is *not* a novel idea, the pipeline is used by **Netflix** (since 2010) and **Pinterest** (since 2013) to just name a few. Pinterest manages over 30,000 artifacts with the pipeline. We're not only using open source toolset but industry standard workflows as well, it's a win-win.

## Terminology

Let's get a couple of terms out of the way so we can get to the good stuff.

**Build** pipeline refers to a process that takes an ISO from upstream repo (Ubuntu, CentOS, GCP image, AWS AMI, Docker image) and create an image that could be used to provision a bare metal server, VMware instance, GCP, AWS, or Docker container. This can also be referred to as build artifact, machine image or simply **Artifact**.

**Provision** pipeline takes the build artifact and creates the resource on a physical server using Foreman, on VMware and GCP using Terraform.

Generated artifacts are uploaded to a **Registry** so they can be versioned and maintained. This is similar to Puppet Forge or Docker Hub.

## Toolset

The goal of this pipeline is to use open source tools. Other than a few bash scripts to glue them together there is no homegrown solution to maintain or onboard people on.

**Jenkins** is at the heart of it all. It's used to generate Packer templates, build, test and upload artifacts to Registry. When it comes to Terraform it can also test, plan and apply resources.

Okay, so I mentioned a few more tools in there, I'll explain.

**Packer** is the tool Jenkins runs to pull the bare ISO from upstream and generate an artifact. Artifacts can be generated for VMware ISO, AWS AMI and any other cloud providers out there.

**Terraform (TF)** is Puppet for API driven infrastructure. Its code written in JSON that essentially allows us to provision resources in VMware or GCP. Its cloud agnostic similar to how Puppet is OS agnostic.

**Consul** is required during the provisioning pipeline. While most of the settings can be configured during build process some things like hostnames or database settings based on data center or region can only

be configured after a machine has been provisioned. It also doubles as an asset registry and key/value store.

**Vault** uses Consul key/value store as a backend to store secrets. These cannot be hardcoded during the build process for security and other reasons. In fact both Consul and Vault servers can run on the same instance.

**Foreman** is used to provision bare metal nodes in physical datacenters. TF can only provision API driven infrastructure so technically we can use Foreman API as there are a few third party providers available.

**Kubernetes** is used to orchestrate containers. While Docker is the Engine that the container runs on each container has no knowledge of other containers in the ecosystem, this is where orchestration comes into play. Orchestrators like Kubernetes can launch and maintain multiple containers, aka pods.

These are just a few of the tools but of course you need something like **Puppet** to configure OS and **Icinga2** to monitor this infrastructure.

## Concepts

### Baking and Frying

- Baking is an artifact purpose built for a specific application.
- Frying is a generic artifact that could be used across many applications.

This is best illustrated using an example. A while back we had to test puppet code locally on a Vagrant machine. Every time we were spinning up the VM it would install packages and pull dependencies on the fly, this is **Frying**. It used to take around 7 minutes for every new VM to spin up and this was getting quite tedious. To get around this we used Packer to “bake” a new Vagrant box with Puppet. After **Baking** the new VM which now has all the baked goods spun up in 6 seconds. As an added benefit it was stable and reusable.

If you notice we said Baking and Frying, not vs. You don't have to choose between one or the other, pick what works best for the

application and tune accordingly. Generally Baked images are more stable compared to their Fried counterparts.

## **SLA Inversion**

The build and provision pipelines are dependent on various pieces of infrastructure that would in turn determine the SLA of the pipeline. For example when building an artifact the availability of BitBucket, YUM Repos, Puppet, Jenkins, DNS, etc. determines SLA for the build pipeline while Foreman, GCP, AWS, VMware availabilities give us SLA for provision pipeline.

Read more about [SLA inversion](#).

## **Infrastructure as Code**

In laymen terms Infrastructure as Code (IaC) is an executable documentation. It lays out instructions which are read by an app to bring up a resource (Bare Metal, VMware, GCP) without actually using any GUI or CLI. If you work with Puppet or Chef this should be a familiar concept. Any change in the infrastructure needs to be codified and tested. Unit, Integration and Acceptance tests are paramount when writing Packer and TF templates.

Read more about [IaC](#).

## Twelve-Factor App

There are many different ways an App can be designed to work with various parts of the Hybrid Infrastructure. One of them is Twelve Factor App design.

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
- And can **scale up** without significant changes to tooling, architecture, or development practices.

### I. Codebase

One codebase tracked in revision control, many deploys

### II. Dependencies

Explicitly declare and isolate dependencies

### III. Config

Store config in the environment

### IV. Backing services

Treat backing services as attached resources

### V. Build, release, run

Strictly separate build and run stages

### VI. Processes

Execute the app as one or more stateless processes

## VII. Port binding

Export services via port binding

## VIII. Concurrency

Scale out via the process model

## IX. Disposability

Maximize robustness with fast startup and graceful shutdown

## X. Dev/prod parity

Keep development, staging, and production as similar as possible

## XI. Logs

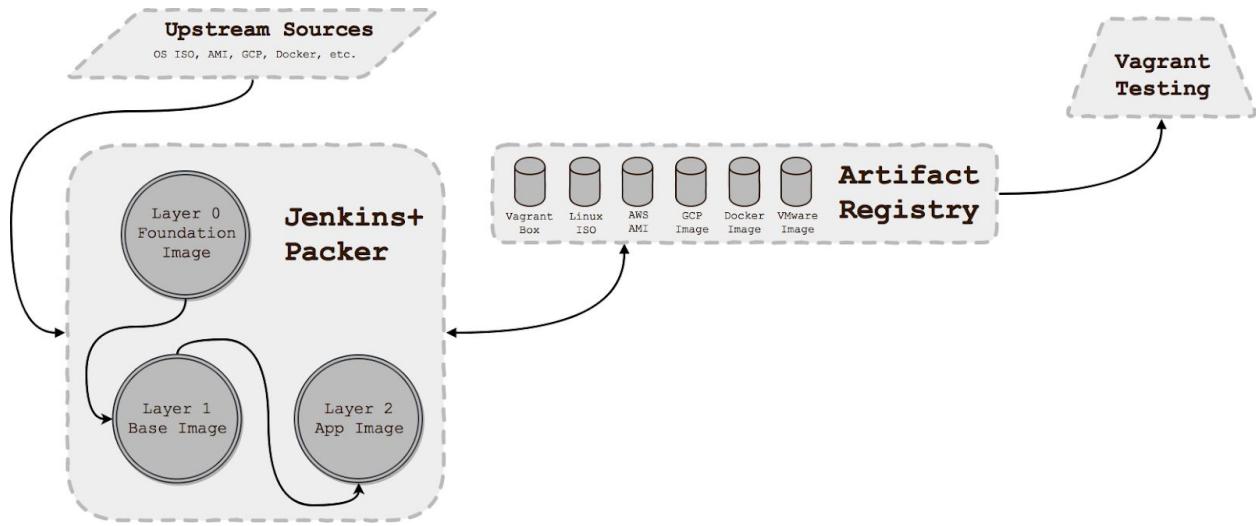
Treat logs as event streams

## XII. Admin processes

Run admin/management tasks as one-off processes

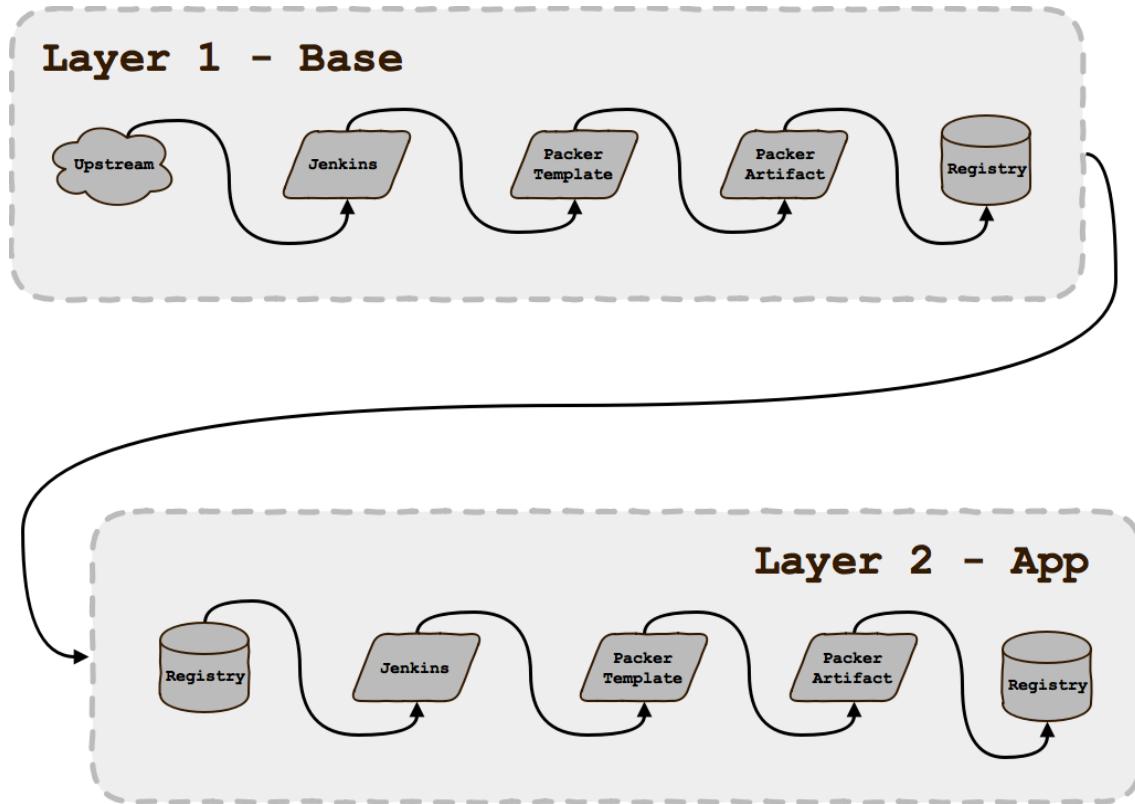
Source: <https://12factor.net>

## Build Pipeline



### Jenkins+Packer workflow

The workflow below highlights the Jenkins+Packer portion of the build pipeline. As you can see Base/L1 can be reused to create App/L2 artifacts. Other than the initial source, Upstream vs Registry, the workflow remains the same.



**Layer 0 (L0), Foundation Artifact:** The Foundation is ISOs released by Canonical, RedHat, Docker and even AWS and GCP. These are bare bones standard linux distribution that can be pulled from any public repo.

Examples would be Ubuntu 14.04 and 16.04 and CentOS 6.8 and 7.2.

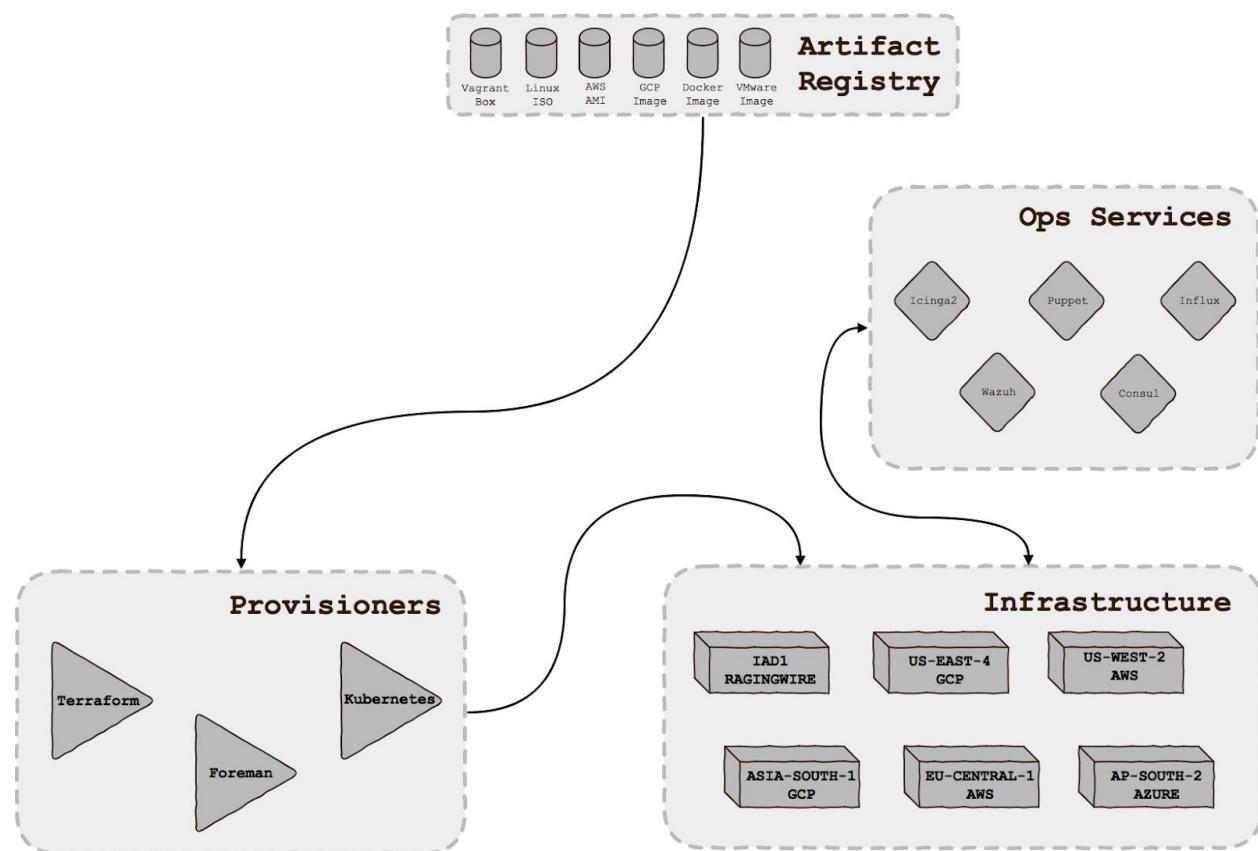
**Layer 1 (L1), Base Artifact:** This is where the org specific operating system with base packages, monitoring and security best practices installed. These are shared across all the apps. Examples would be Puppet, Icinga2, Logrotate, etc. We should not have more than a handful of these, these artifacts are usually built and maintained by the Infrastructure team.

**Layer 2 (L2), Application Artifact:** Configuration, Packages, GPU drivers specific to an app. Examples could be Data Science, Web, Redis, Mongo, Memcache, etc. Base artifacts (L1) can be reused to generate new app artifacts. Don't be too surprised if you end up with a lot of L2 artifacts. These artifacts could potentially be created by Devs as well so they can test out their docker container with Vagrant before being used in production.

Note: Generally the ratio between L0 and L1 should be 1:1 and app artifacts (L2) should not be reused to bake newer artifacts (L3 or higher) since the dependency chain would become unwieldy.

Time for another example, let's take the SSL exploit, we've had a few of those lately. If a L1 artifact is vulnerable to an exploit any L2 artifact created off of that is vulnerable as well. Assume you have L3 artifact so that's another layer that has to be patched. This might not be a problem if you have a few but this pipeline has the potential to generate thousands of artifacts, try figuring out that dependency chain. Another example would be Kernel security patches. If L1 is vulnerable any L2 artifact built off of that is vulnerable as well and requires to be patched.

## Provision Pipeline



**Artifact Registry** stores artifacts built by Packer for Bare Metal, VMware, GCP, AWS, Docker, etc. It can be used to retrieve L1 images to

build L2 images and keep track of various dependent images and their versions.

**Provisioners** pull artifacts from Registry and use the metadata to provision VMware and GCP resources. Using TF will allow you to codify and version control the infrastructure. Foreman comes into play when you need to provision hardware nodes. We can manage Foreman via TF with any community providers out there.

**Ops Services** are various services used by all nodes in a particular site or region. Important ones being Puppet, Icinga2, Consul and Wazuh to just name a few. It might not be clear in the flow chart but each site/region needs to have some of these resources operating independently in case of an intersite outage, like Consul and Puppet.

## Infra vs. App Environments

Infrastructure goals are different from Application. From an Infrastructure perspective all environments, even development, are “production” critical. If Development or Stage App environments go down it affects developers, so they are considered “production”, at the same time Infrastructure needs to be tested before being deployed into production.

We achieve this by separating our environments into distinct ones.

### Infra Environments

- Sandbox
- Canary
- Live

### App Environments

- Dev
- Stg
- QA
- Prd
- Ops

Each Infra environment could have all (or some) App environments, the hierarchy would be as follows:

- Sandbox
  - Dev

- Stg
- Canary
  - Dev
  - Stg
  - QA
  - Prd
- Live
  - Dev
  - Stg
  - QA
  - Prd

This way we can actually apply production changes in a Canary environment before applying to the Live environment.

## **Hostname Format**

The current hostnames are designed more for unicorns than cattle. If we want to start treating resources like cattle the hostname should give just enough info the type of application is running on at the same time be as random as possible.

One suggestion would be to use the following format:

ROLE-UUID.APP-ENV.INFRA-ENV.REGION.fqdn.com

Examples:

- puppet-server-f9o4hs83.dev.canary.iad1.fqdn.com
- base-web-10df54h7.prd.live.us-east-1.fqdn.com

## **Can we add another cloud provider into the mix?**

Absolutely. Let's take AWS for example; Packer can build AWS artifacts (AMIs) and TF can provision AWS resources. Let's take another example, Azure; Packer can build Azure artifacts and TF can provision Azure resources. We can keep doing this all day.

## **Outstanding questions**

- How do we keep the registry clean and without cruft?