Containers and Pivotal Cloud Foundry

By Saurabh Gupta

Pivotal

WHITE PAPER

Table of Contents

Introduction

Containers have become popular with developers and operators alike because they offer a convenient way to achieve benefits that were hard to obtain otherwise:

- Consistency of deployment
- Consistency of execution
- Easing the Developer-to-Operator handoff

Similar to virtualization, where hypervisors and machine images are not a full virtualization platform, container technology is one piece of a larger set of responsibilities needed for an enterprise platform. Developers and operators have found that simply taking an image and creating a workload out of it as a container is not the "full" solution: i.e. "containerizing a workload" does not automatically give operators and developers the suite of features they would like with regards to things like high availability, auto scaling, traffic routing, logging, etc.

An "Enterprise Platform" is the term of trade used to describe the system that provides all these capabilities on top of the basic function of running a workload as a container. Many more things are needed before container images can be run at "enterprise grade": i.e. with capabilities such as security, high availability, unified logging, application lifecycle management, infrastructure automation, etc.

Since the core notion of containers is devoid of any notion of the capabilities of a Platform, enterprise operators are left having to bring these capabilities in on their own, either by writing their own tools, or by pulling together a disparate set of tools, utilities, and products at varying levels of maturity from other sources and suppliers.

Containers and Docker

Docker, with its developer friendly user experience, has dominated the conversation about containers. "Docker", as a term, implies many things:

- a commercial venture-backed startup company: "Docker, Inc"
- an open source container project without a foundation governance model "Docker project"
- the brand for a commercial product produced by Docker, Inc.
- the name often used for the popular open source runtime "Docker Engine"
- a container image format known as a "Docker image"
- a developer desktop tool-chain to simplify development and packaging "Docker for Mac & Docker for Windows"
- a short-hand word used to represent container technology

However, In the context of this document, "Docker" implies only the container image format.

Containers and Platforms

As noted above, a container image format does not make a "Platform." A Platform is a suite of capabilities that work together to offer developers and operators a simple way to deploy, monitor, manage, scale, secure, and connect applications.

Containers, as detailed in this document, are processes running on an operating system. In of itself, container technology is unable to provide all these features. However, container technology is an important part of an enterprise platform: it is central to how applications are deployed and managed, but the systems that provide these capabilities are additive on top of the core container technology for running applications.

Containers and Pivotal Cloud Foundry

Pivotal Cloud Foundry (PCF) is a complete platform that takes payloads from developers (either as compiled artifacts like jar and war files, or as pre-built container images) and provides a complete system to schedule these payloads, and provides all auxiliary services of a platform indicated above (load balancing, high availability, auto scaling, unified logging, etc.) The payloads can be delivered via any means (directly by developers as dev payloads, and via continuous integration/continuous delivery (CI/CD) pipelines for quality assurance (QA) and Prod level deployments). PCF utilizes containers extensively to provide its services and keep its payloads up and running. However, containers are only a part of the system. PCF orchestrates containers with ease, but that alone is not what makes PCF a platform. A lot of sub-systems act together to coordinate, monitor, and support these containerized payloads.

Scope of This Document

This document takes the reader through the following topics:

- **Containers:** a high level introduction to level-set the information about containerization and create a common knowledge base from which the rest of the document draws upon
- **PCF and Containers:** how PCF uses containers, both those that are supplied as pre-built images and those that PCF builds itself using buildpacks
- **Buildpacks:** details about buildpacks and how they work
- **Similarities:** the similarities between the two approaches that PCF takes in container deployment
- **Contrasts:** the differences and tradeoffs that occur between one approach and the other.

This document is not a Docker or PCF tutorial, nor is it an architectural deep-dive on these topics (although relevant architectural details are presented where needed). It gives a high level overview of container technology and how containers are leveraged by PCF. This document narrowly focuses on how PCF uses containers, including the use of "upstream" technologies like Docker images.

Acknowledgements

Deep thanks to the stellar team at Pivotal for their feedback, commentary, and suggestions: James Bayer, William Pragnell, Onsi Fakhouri, John Feminella, Raghvender Arni, Richard Seroter, Ian Andrews, David McClure, Adam Goldberg, Vinod D'Souza, and James Williams.

Containers

Containers have existed in the *NIX ecosystem for many years. Pivotal Cloud Foundry has used containers at least since 2011 to provide its application orchestration and management capabilities. Recently, Docker has created a user-friendly developer-centric system for containers. This has lead to an explosion of interest in using containers for ongoing projects and initiatives. However, “Docker”, in many cases has become synonymous with “containers” (just like Kleenex for tissues or Xerox for copying). The fact that Docker is an image format, a product, and a company causes a lot of confusion when the context is not clarified.

Generally, “Docker” as a product represents ~35 different products (e.g. docker-swarm, docker-machine, docker-bench, etc.), all at varying stages of maturity and operational readiness. Within the context of Pivotal Cloud Foundry, “Docker” implies the docker image format that organizations can use to package applications; PCF then puts this image into a process and runs it with a set of guarantees about its behavior—that is, a container running a Docker image. Once the Docker image is instantiated as an application process on the PCF platform, it is like any other PCF-managed application instance.

The level of awareness and understanding of containers varies widely. This section provides a short primer on containers to level set readers on the basic concepts around containers and seeks to demystify the topic to newcomers

“It’s a Process”

A container running on a machine is just a process on the host operating system like any other. The unique part with respect to ‘containers’ is in how that process is isolated and restricted in very intentional ways.

When you run a command on a Linux machine, it runs with certain permissions and for a certain duration. For example, the **`ls`** command will (at a very simplistic level) do a directory listing:

- It will run for the duration of listing the directory’s contents
- It will have a process id (PID) for the duration that it is alive
- It will only be able to see what the user has access to (e.g. if the user running the command doesn’t have ‘read’ access on a directory, like the home directory of another user, **`ls`** will be unable to show the user its contents)

In the above example, **`ls`** is a ‘read only’ operation: nothing is changing about the state of the file system when this command is run. On the other hand, something like **`cp`** (copy) is a read/write operation: something has to be read and then written somewhere else. If a user doesn’t have write permissions on the target directory, the command will fail. If the user has write access, the command will succeed and the state of the file system will change. While it is executing (whether ending in success or failure), the **`cp`** command will also have a **`PID`** associated with it.

Similarly, a "container" is just a process with additional isolation properties supplied by features of the operating system: one that is run by a special user that has a certain set of permissions and can access a certain set of file systems and other resources. The three core underpinnings of containers in Linux are **cgroup**, **namespaces**, and **filesystems**.

cgroup

"cgroup" is an abbreviation for control group. As defined in the Linux spec[1]:

> cgroup is a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner.
>
> ...
>
> cgroups form a tree structure and every process in the system belongs to one and only one cgroup. All threads of a process belong to the same cgroup. On creation, all processes are put in the cgroup that the parent process belongs to at the time. A process can be migrated to another cgroup. Migration of a process doesn't affect already existing descendant processes.
>
> Following certain structural constraints, controllers may be enabled or disabled selectively on a cgroup. All controller behaviors are hierarchical - if a controller is enabled on a cgroup, it affects all processes which belong to the cgroups consisting the inclusive sub-hierarchy of the cgroup. When a controller is enabled on a nested cgroup, it always restricts the resource distribution further. The restrictions set closer to the root in the hierarchy can not be overridden from further away."

There are cgroup controllers for CPU, Memory, I/O, Network, PIDs (Process IDs) and Devices. For each, the cgroup allows control of various parameters such as weight, max, etc. that control the allocation of a resource to a process belonging to the cgroup.

The **CPU** cgroup controller:

- Regulates distribution of CPU cycles based on the specified 'weight' of a certain cgroup expressed in units of CPU shares
- Tracks CPU usage by a process/user

The **memory** cgroup controller:

- Tracks page cache and anonymous memory (Userland)
- Enforces the hard limits on memory availability and triggers out-of-memory errors in processes that attempt to exceed it

The **I/O** cgroup controller:

- Regulates the distribution of I/O resources using weight-based and absolute bandwidth/IOPS limit distribution
- Enforces the IOPS limits for read bytes per second, write bytes per second, max read I/O ops per second, max write I/O ops per second

1. Content for this section referenced from: https://github.com/torvalds/linux/blob/master/Documentation/cgroup-v2.txt

The **network** controller:

- Controls network traffic priority and traffic class
- Controls egress by tagging packets based on the sender cgroup and assigning it a priority

The **PID** controller:

- is used to allow a cgroup hierarchy to stop any new tasks from being fork()'d or clone()'d after a certain limit is reached
- Allows a hierarchical control of PID resources, with the most stringent policies being propagated downwards

The **devices** controller:

- Controls access to specific mounted devices on the host (/dev/*)
- Can be used to give containers access to host resources, such as GPUs

Namespaces[2]

While cgroups control what a process is able to do, namespaces control what view of the VM a process is able to have. Namespaces allow a process to treat a global resource as if the process had its own isolated instance of that resource. Linux provides six namespaces:

- **IPC:** isolates resources like the POSIX message queue
- **Network:** isolates network devices, IPv4 and IPv6 protocol stacks, IP routing tables, port numbers, etc.
- **Mount:** isolates the set of file system mount points, meaning that processes in different mount namespaces can have different views of the filesystem hierarchy
- **PID:** isolates the process ID number space, meaning that processes in different PID namespaces can have the same PID. PID namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs
- **User:** isolates security-related identifiers and attributes, in particular, user IDs and group IDs. A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace
- **UTS:** isolates two system identifiers: the hostname and the NIS domain name

A namespace is alive as long as it is being used by a process running inside of it or there are mounts inside of it. When the last usage goes away, the namespace is destroyed.

When a process starts, it is in one namespace of each type, so it is able to get an isolated view of its resources from all six of these dimensions.

2. This content referenced from the Ubuntu Linux documentation http://manpages.ubuntu.com/manpages/wily/man7/namespaces.7.html

File Systems

Now that Linux can control what a process can do (cgroup) and what a process can see (namespaces), it can control the file system the process has access to by mounting it in the namespace of the process. Additionally, the process can have read-only access to most of the file system and read-write access to a small part of it.

These file systems are at the core of the concept of an "image": an image (e.g. a Docker image) is just a set of serialized file systems with some configuration and metadata. When an image is deployed as a container, the container process gets the image's file systems expanded as a mounted file system in its namespace.

Images and File Systems

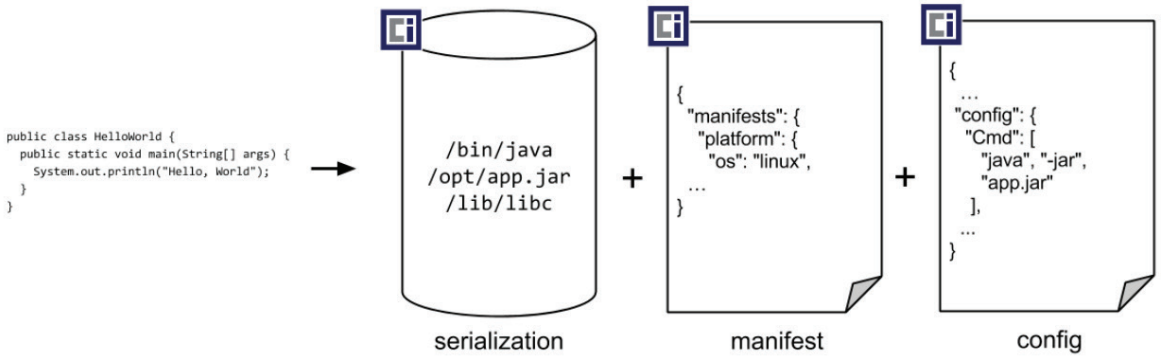
The Open Container Initiative (OCI, https://www.opencontainers.org), of which Pivotal is a member, has created a clear specification around what an image should be. You can liken this to a standards body setting a standard for JPEG or PDF files.

An OCI image is made of 3 things[3]:

1. **Image Manifest:** contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more file system serialization archives that will be unpacked to make up the final runnable file system
2. **File system Serialization:** the actual serialized bits of the file system referenced in the image manifest
3. **Image Configuration:** includes information such as application arguments, environments, etc.

The above items are illustrated here:

The contents of an OCI image, sourced from the OCI spec on GitHub



In common parlance, when you are "building a container," you are actually building an image with a particular format. What is commonly referred to as a Docker "container," therefore, is an image that follows the Docker image format. Docker Hub is a registry of these images.

3. Content referenced from OCI spec at https://github.com/opencontainers/image-spec

runC

Now that we have an image, and we have a process structure to run it as a container, we need something that can make that happen. That "something" is a runtime.

Just as OCI has an image format specification, it also has a specification on how to run the image once it's serialized file system bundle has been unpacked on disk[4]. A runtime implementation that follows the OCI Runtime Spec will run an OCI container. Various tools can surround the runtime that do the auxiliary tasks of downloading an OCI Image, unpacking the image into the filesystem bundle, and then invoking the runtime to run the container. In this way, the OCI runtime is decoupled from the actual image: OCI runtimes can run a container without using a container image as a source.

runC is the reference implementation of the OCI runtime spec. It is a CLI tool for spawning and running containers according to the OCI specification.

Some container runtimes, such as Docker and PCF's Garden, use runC. In fact, Pivotal joined with Docker last year to help standardize that runtime as part of the Open Container Initiative.

Ultimately, as long as the runtime conforms to the OCI specification, the actual underlying implementation details of a container runtime should be immaterial to anyone looking to use container technology. The focus of PCF is to be a platform that runs containers in an isolated way with strong guarantees about their behavior and without making developers and operators explicitly bound to a particular OCI spec implementation. PCF's abstractions, and features such as pluggable backends, help achieve that separation of concerns.

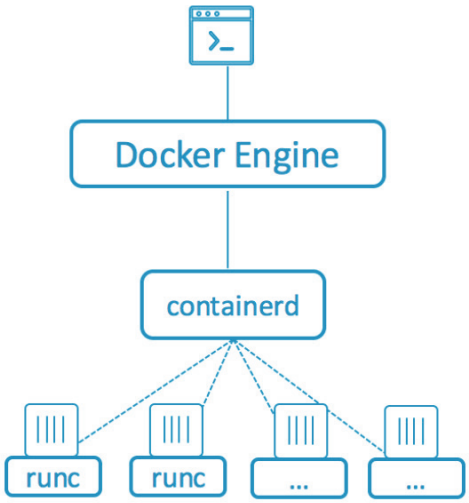
An illustrative analogy is that there are a multitude of things that can make and read PDFs, but PDF viewers should be decoupled from the software that created the PDF. It's not desirable to have a PDF Viewer that says "I refuse to open PDFs unless they were generated by Adobe Acrobat v7.2." PDF is the underlying standard; Acrobat is just one runtime implementation. What matters is the standard and not the runtime, which is an implementation detail. In the context of containers, Docker images are an image format, and there are many runtimes that support running those images.

Details of how PCF works with containers are given in the following sections.

Running runC

runC is a simple command line utility. It runs on the host. Generally, these hosts are part of a larger cluster. The cluster manager would like to issue commands to these hosts (nodes) to do different things with containers. Essentially, the cluster manager would like to issue commands to runC remotely. But runC is a command line utility and has no remote or REST interface. Therefore, all that extra wrapping around external communications, security, etc. has to be done by some other tool. These tools can be open source or proprietary. Docker's stack uses Docker Swarm as the cluster manager, and Docker Engine (built on containerd) as the runC manager on the node:

4. This section is referenced from https://github.com/opencontainers/runc



Same Docker UI and commands

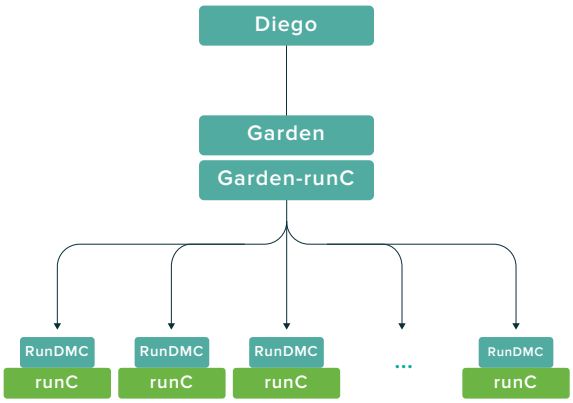
User interacts with the Docker Engine

Engine communicates with containerd

containerd spins up runc or other OCI compliant runtime to run containers

Source: https://blog.docker.com/2016/04/docker-engine-1-11-runc/

Similarly, PCF uses an enterprise grade clustered container scheduler known as Diego, which orchestrates the container runtime on many hosts, each with Garden-runC which manages all of the containers running on one host. More details on how Garden works are given in the next section.



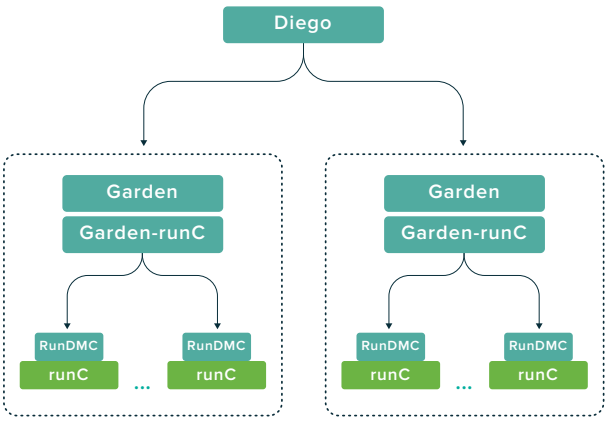
Diego issues commands

Diego interacts with the Garden API

Garden uses the Guardian pluggable back end for runC

RunDMC wraps runC to create and run containers

Additionally, since PCF also clusters the hosts to create a scale-out architecture, the schematic above applies uniformly to multiple hosts:



Diego issues commands

Diego interacts with the Garden API

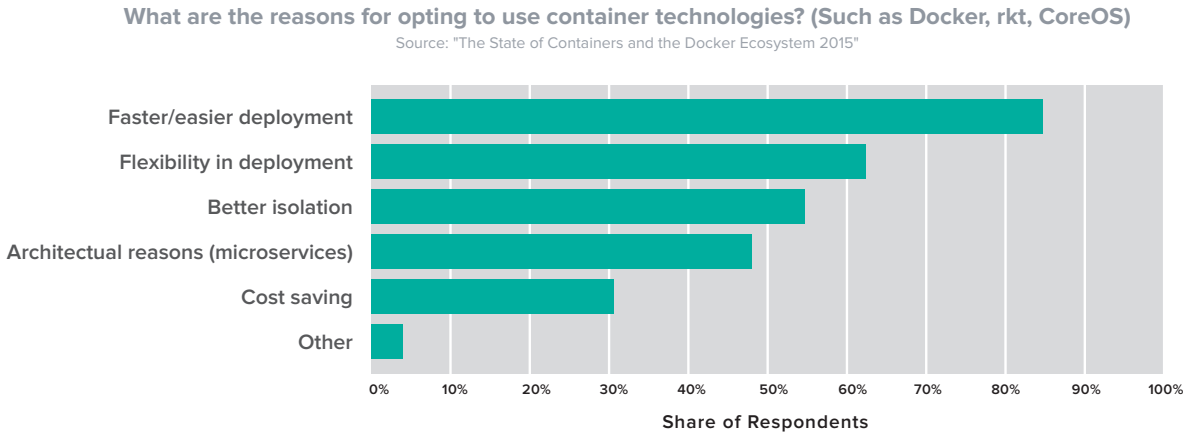
Garden uses the Guardian pluggable back end for runC

RunDMC wraps runC to create and run containers

PCF and Containers

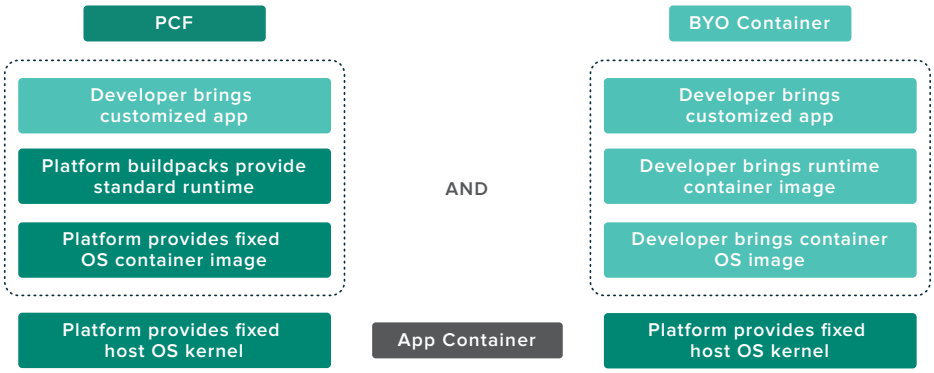
Cloud Foundry has used containers from the very beginning (2011). The original rationale, which still holds true today, was anchored in two core tenets:

1. The ability to better utilize the underlying infrastructure
2. The ability to provide consistent, production-ready, resilient environments for developers in a highly productive and easy-to-use manner



Containers get deployed onto PCF in one of two ways:

1. **Pre-Built Images (e.g. OCI based Docker):** where a developer packages all their dependencies through something like a Dockerfile to build their image themselves, and publishes it on a public or private registry, and then instructs PCF to deploy, orchestrate, and run that container within PCF.
2. **Platform Built Containers via Buildpacks:** where a developer brings just the code/compiled artifacts to PCF. In this case:
    a. PCF selects the appropriate buildpack which contains specifics of the dependencies for that runtime, as defined by compliance and platform operators.
    b. Packages in the dependencies (such as the correct runtime and middleware) and makes the package immutable. This package is called a Droplet: it is a "proto-image" in that dependencies are not packaged as serialized layers of the file system, but simply as binaries that will go on a generic base file system layer provided by the platform.
    c. Deploys the droplets on demand by streaming the droplet to the node, and orchestrating the lifecycle of the container process.



The Pre-Built + Platform Built (or, more pragmatically, Docker + Buildpack) approaches together give developers and enterprises more flexibility and choice. There is a brief discussion in the Buildpacks chapter about how to decide which approach to use in different situations.

Pluggable Backends

PCF has the concept of "Pluggable Backends." Container runtimes (such as runC) are generally command line tools and do not have remote communication features. PCF's remote communication components have a front end: one that implements an API to communicate between the node and the cluster manager, and a back end: one that mediates communication between the front end and the container runtime.

The cluster manager (Diego Brain) communicates with the cluster nodes (Diego Cells, which talks to the Garden components on the host). Garden's pluggable backends then translate the instructions sent to the runtime on the node, and use the runtime to run the payload.

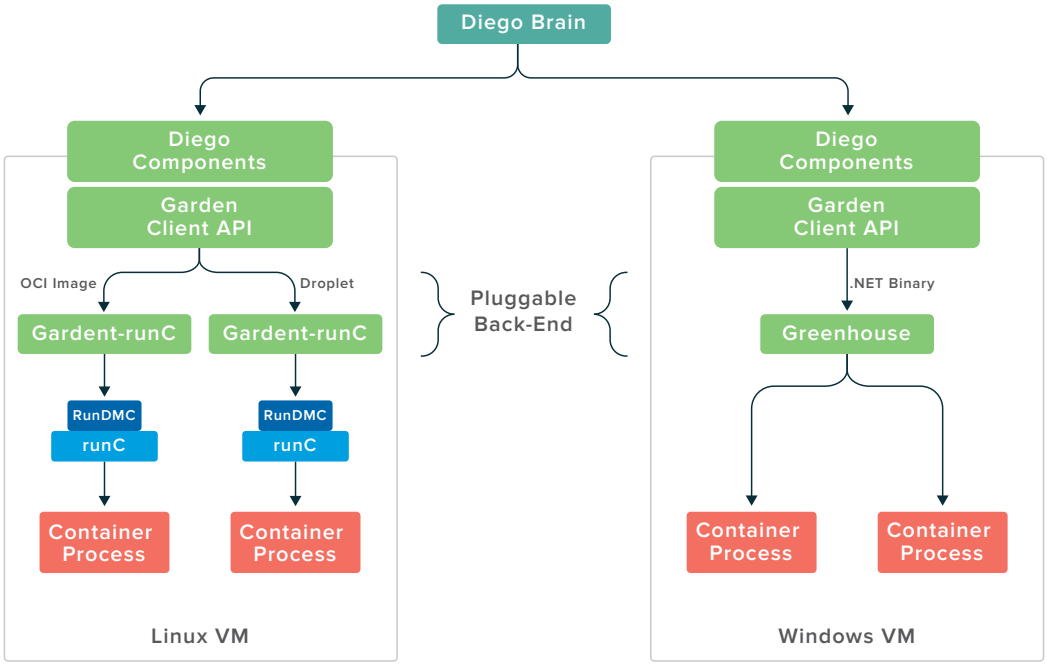
In the case of an OCI image (i.e. a "Docker Image"), the Garden-runC backend will invoke the runC libraries to initialize and start the container process based on the image.

In the case of a Droplet, since runC doesn't have to do any of the unpacking and file system deserialization, Diego will unpack the droplet onto the container file system (which is the pre-packaged rootfs[5] built into the platform) and start the container process.

Also, the pluggable back end system allows other operating systems and runtimes to be easily incorporated into PCF: it is not bound to a certain inbound payload. This capability is used to create containers on Windows machines for running .NET apps using Windows concepts equivalent to cgroups and namespaces.

5. rootfs is further explained here: https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt

The pluggable back end system can be represented as follows:



Remember, a "Container" is a Linux process with cgroups, namespaces, and filesystems. There is nothing proprietary about it. A "Docker Container" is simply a shorthand for saying "take a Docker image and make a Linux process out of it with the correct cgroups/namespaces and mount the file systems in the image." Once a container process is running, it is just like any other process on the host. This means that the process doesn't really "care" if it was created from Docker image or a Droplet. It will run the same.

Platform Opinionation

PCF approaches container orchestration in an opinionated manner, by making certain choices around what capabilities to support to ensure that the platform can achieve its overall goals for security, elasticity, flexibility, and standardization.

Docker's Dockerfile instruction set provides many features to configure container behavior. However, not all of these instructions are conducive to creating a multi-tenant, secure enterprise platform, as we'll discuss below.

To that end, PCF is continuously evolving its support for various instructions in the Dockerfile. For instance, PCF controls the ports that a container can expose, to prevent malicious code in a container from connecting to other containers, or reaching out with unsupported protocols, and for keeping inbound network traffic compliant with the platform's routers' capabilities (especially with regard to things like load balancing, HTTP sessions, etc.).

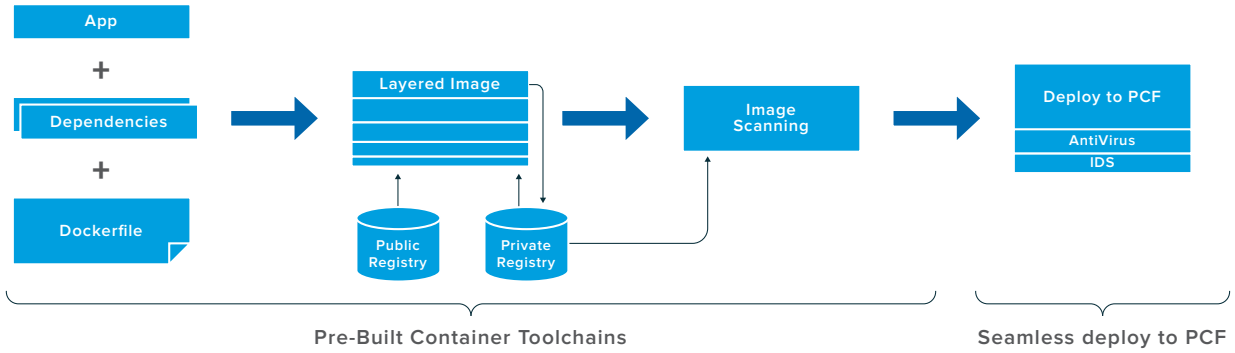
Some Dockerfile instructions work great in development scenarios, but are less suitable for large scale, highly distributed, multi-tenant, secure platforms. The PCF team is working to bring these features to the full platform. For example, as of this writing (Q4, 2016):

- **Volume:** currently, this is a roadmap item. Custom mounted volumes can be problematic in case of VM failure: there is significant bookkeeping required to track which VMs had which external volumes mounted for which containers, and recreate those mappings if either the container or the host VM fails. Additionally, the system also has to account for external volume failure and handle those situations gracefully.
- **Expose:** multiple port exposure is a roadmap item, largely for reasons explained above.
- **Healthcheck:** custom healthchecks add more configuration overhead to deployment manifests and can create security vulnerabilities. However, a 'safe' way of doing this is being explored as a roadmap item.
- **User:** specific UIDs are irrelevant in PCF where the platform is dynamically creating containers on host VMs. Processes can make no assumptions about the users present on the host machine, since these users are created programmatically with the intent of creating the correct container configuration.

In general, PCF is making defensive decisions about the type of access containers should have to the surrounding system. In some cases these opinions can conflict with a developer's experience with running Docker containers locally. However, when evaluated in the context of a platform, these opinionations attempt to make the best possible trade-offs and are continuously evolving.

Docker (Pre-Built Image) Scenario

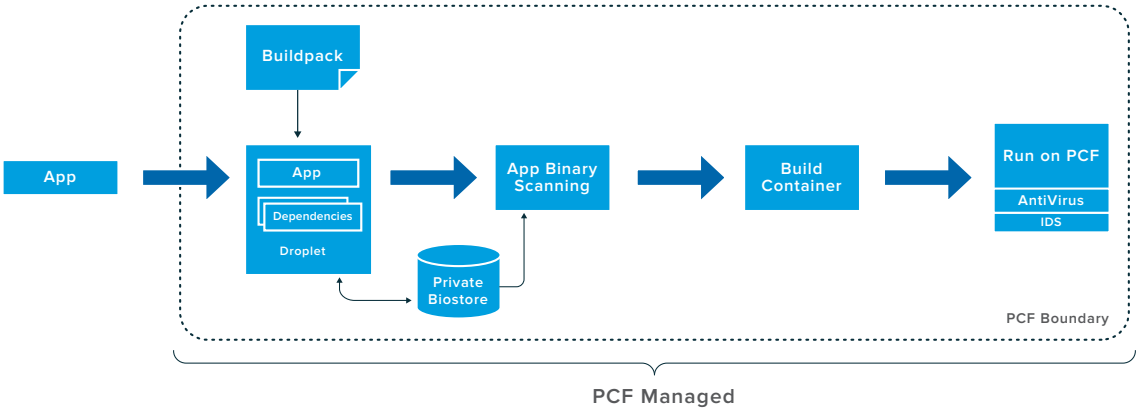
A typical deployment pipeline to PCF using existing Docker image creation toolchains might look like the following:



In the above scenario, developers pre-build their images before deploying them on PCF. App developers keep track of all the dependencies their application will require and get the base images that already contain the dependencies, or package them in with their application when the image is built based on instructions in the Dockerfile. App developers leverage the existing image scanning toolchains that exist within the enterprise before the image is deployed into PCF. PCF then manages and orchestrates the container based on that image just as if it had built the container itself.

Buildpack Scenario

A typical deployment pipeline in this scenario might look like this:



In the above scenario, the app developer hands off their compiled code (after putting it through a CI/CD pipeline, if needed) to PCF. PCF then:

- utilizes the buildpack, which is a 'corporate standard' way of managing dependencies to bundle in the "right" middleware and runtime
- runs scans on the code binaries to detect any potentially out of compliance libraries or licenses, using a framework like AppDog (for Java)
- builds a droplet that contains the application binaries (e.g. .jar or .war files for Java) and runtime binaries (e.g. the JDK for Java)
- the droplet is sent to the different nodes that have been selected to run it, via the auction process
- the Garden-runC backend at each node will build the container and deposit the droplet in the node and start the process

Buildpacks

Before we get into the details of buildpacks, let's take a step back and really think about the reasons why containers are all the rage today.

Historically, developers and operators have struggled mightily to create a consistent, replicable runtime environment. Most issues in deploying software and running it reliably stem from inconsistencies between how an application was built during development, and how it is put together for deployment.

As we have seen earlier, in a way, all processes are containers. The parlance of "containers" today explicitly and very precisely implies that a process is created with very well defined cgroups, namespaces, and file systems, such that its behavior can be very tightly controlled. "Containers" are all the rage today because, for the first time, there exist tools and capabilities that allow these containers to be created in a repeatable reliable manner.

The question then becomes: how to create this "container" process? There are two ways this process can be created:

1. **Buildpacks:** Have a well defined algorithm that takes minimal input and depends on automation, standards, and supported components of a platform to create a process. The repeatability and reliability come from the automation and standards enforced by the platform and applied uniformly for all payloads.
2. **Images:** Have a pre-baked image that contains all the dependencies, and run it as-is. The repeatability and reliability then come from how the image gets built rather than how the container is created.

Both are valid approaches, but each has trade-offs.

How Buildpacks Work (Source-to-Container)

When a developer presents a payload to PCF, PCF has to make a decision about how to compile and package that payload into a deployable artifact. The buildpack contains all the logic that PCF uses to decide what to do with an application payload that has been presented to it. The buildpack is a set of scripts that can contain any logic needed to fulfil their function.

A buildpack is a set of three scripts in a 'bin' directory that run in the following order[6]:

1. **`bin/detect`:** this script determines whether or not to apply the buildpack to an application. The script can use logic such as detecting certain files, or directory structures in the application payload to make the determination.
   This script takes the directory of the payload as an argument returns an exit code of 0 if the buildpack can be used with the application.

6. This section references Cloud Foundry buildpack documentation (https://docs.cloudfoundry.org/buildpacks/custom.html)

2. **`bin/compile:`** this script takes the root directory of the application and the target directory of where to put the output of the compile process. Once an application succeeds with the detect script, this script packages the application's dependencies as specified in the buildpack, and builds the application's droplet which can then be deployed in a running container.
3. **`bin/release:`** this script provides feedback metadata to Cloud Foundry indicating how the application should be executed. The script is run with the directory of the application as an argument, and in response, generates a YML file that lists the commands to be run to execute the payload once it has been deployed.

Buildpacks are stand-alone entities. They have no knowledge or awareness of concepts like images, containers, or Cloud Foundry. It is even possible to run buildpacks independently outside of PCF as part of any CI/CD pipeline as well.

PCF utilizes buildpacks as a foundational concept to ensure that all inbound application payloads get the exact same treatment. In PCF, buildpacks can only be set by platform operators, so they greatly strengthen the compliance aspect of bringing payloads onto PCF.

In this way, buildpacks give PCF a "source-to-container" capability that lets organizations use their CI/CD pipelines to convert checked-in code to a functioning container on the platform.

"Source-to-Image" (s2i) compared with Buildpacks

Recently, tools like OpenShift have talked about "source-to-image" systems where a toolkit is provided to convert checked-in code into images via a standardized process[7]. Just like buildpacks, s2i requires the following scripts to be provided:

1. **`assemble:`** builds and/or deploys the source
2. **`run:`** runs the assembled artifact
3. **`save-artifacts:`** is an optional script that can be used to capture artifacts for incremental builds
4. **`usage:`** is an optional script that displays usage information

S2i leverages the above scripts to build images as follows (from s2i documentation):

1. s2i creates a container based on a builder image (from a Dockerfile), and passes that container a tar file that contains the application source and any other incidental build artifacts (in the case of incremental builds)
2. Sets the environment variables
3. Starts the container and runs the **`assemble`** script, and waits for the container to finish the primary process
4. Commits the container, setting the CMD setting for the output image to be the run script
5. Tags the created image with the provided name

7. This section references the s2i documentation: https://github.com/openshift/source-to-image

It can be seen that the s2i process is very similar to buildpacks, but in some cases, a bit different:

- Both are a series of defined scripts with defined inputs that produce deployable artifacts: buildpack processes produces a droplet, while s2i processes produce an image.
- While it is an implementation level detail:
  - Droplet construction takes place within PCF, and the process to build a droplet runs as a container on one of the Cell VMs.
  - Image construction on s2i is an OpenShift-specific process where the container process to build the image runs outside of PCF, but PCF can run the image that is the output of the s2i process.
- With buildpacks, developers only have to provide their code, but with s2i, there is greater onus on developers to provide more details around the deployment of the code itself.

Choosing Buildpacks or Images

Often, the question arises around when to use buildpacks and when to use images for deploying software. This sections aims to provide some outlines for when each might be most appropriate within the context of PCF.

Using Buildpacks

Buildpacks take their input very close to the actual source code. Depending upon the runtime, the buildpacks either take source code (such as for Python or Ruby), or take compiled binaries (such as for Java).

Therefore, buildpacks are most suitable for cases where some of these hold true:

- Source code for the item to be deployed is available and is to be deployed in a runtime for which a buildpack exists or can be created
  - In the case of a net-new project, this can be an obvious choice, since developers are writing code from scratch and can design for modern applications and patterns such as microservices and 12-factor applications
  - In the case of legacy applications, developers have a choice to update the code base to make the applications more modern and introduce concepts like externalized configuration, etc. and deploy the code using buildpacks
  - In case of legacy applications where source code is available but there is no appetite for updating or modernizing the code base, these applications can still be deployed using buildpacks after some minimal changes such as removing any vm-based dependencies (to permit seamless container orchestration)

Using buildpacks, in many cases, allows developers to move away from proprietary runtimes into more generic, open source runtimes. Having access to the source code also allows developers to tie changes to a CI/CD pipeline that can respond to code checkins and run a deployment pipeline in a zero downtime manner.

Using Images

Developers can choose to use images in cases such as:

- Source code for the application is not available, such as a legacy application where the development team is no longer with the organization
- Commercial applications that are delivered as binary bundles
- Applications that use highly proprietary runtimes or for which buildpacks are not available and/or cannot be easily created

In the above situations, creating a pre-baked image and delivering it for deployment can be a viable solution. However, developers have to take more responsibility for complying with corporate standards around what can/cannot go into an image, and platform operators have to take on the responsibility of creating internal registries that are kept updated with various patched images, etc.

Similarities Between Image and Buildpack Deployments

There are some significant similarities between the OCI image deployment and the buildpack approaches on PCF. Namely:

- PCF manages either of these payloads equally: fundamentally, it is about getting a process running on a node somewhere
- PCF gives both of these scenarios all the benefits of running on the platform:
  - High availability
  - Failover
  - Unified metrics and logging (see details below)
  - Automatic scaling and traffic routing
  - Application Performance Management
- PCF does a zero downtime update of host OS VMs
- Blue-Green deployments for zero downtime app updates
- Application Security Groups for outbound traffic management
- Providing access to service brokers that allow name-based service binding to external services without recompiling code or rebuilding containers
- Multi-language support (Java, .NET, Node.js, Ruby, Go, PHP, etc.)
- Multi-cloud support (vSphere, OpenStack, AWS, Microsoft Azure, Google Cloud Platform, etc.)
- Spring Framework (ex. Spring Boot, Spring Integration, Spring Cloud Netflix Services, Spring Cloud Dataflow, Spring Zipkin, Project Steeltoe, etc.)

In addition to the items above, there are the following critical areas where deployed applications are treated the same:

Leveraging Platform Add-ons

PCF provides operators the ability to introduce their own agents (e.g. for AntiVirus and Intrusion Detection Systems) into the platform. This improves compliance and consistency in multiple ways:

- PCF attaches these agents to each VM it spins up and manages. This means that all updates, patches, etc. automatically include these agents configured exactly the same way across the whole platform. All managed VMs get these agents, including those that are running containers (any type) and running managed services.
- The absence of configuration drift in agent setup allows the enterprise to have a high quality of assurance and certainty in the integrity of the agents that are running across the platform.
- These agents don't have to be part of the developer workflow and operators can maintain full control over their deployment and use.

Leveraging Container Monitoring and Metrics

A screenshot of PCF Metrics showing Container use history

Since all container processes on PCF are run the same way, they all get the benefit of the common utilities that PCF provides for these processes:

- CPU, Network, Memory monitoring and reporting
  - CPU monitoring for all containers implies that auto-scaling can be implemented for all containers running on the platform: Docker or Droplet, and also .Net containers on Windows VMs
- Log aggregation from stdout and stderr
- Container Process health monitoring to ensure container uptime: if a container process crashes or freezes (detected via a health check endpoint), it can be resurrected from the source image or droplet

Read/Write File Systems

Once a container is up and running in PCF, the underlying file system works the same way, regardless of whether the container was created from an image or via a buildpack+droplet.

The base file system layers stay read-only, and the application is able to write on its own writable layer on top of that.

For all containers managed by PCF, the entire mounted file system is erased when the container is destroyed or stopped. This means running containers only have ephemeral file storage and applications deployed in containers cannot make any assumptions about the file system when they start. All temporary data stored by the applications will be erased when the container exits.

Contrasts Between Image and Buildpack Deployments

When developers choose to build their own images outside of PCF, the onus for managing and maintaining the images falls on the developers and operators. This includes many of the compliance and assurance aspects that would normally be provided by PCF. Some of these contrasts between Docker and Buildpack approaches are:

Developer Considerations

Docker:

- Developers put together images for software they want to deploy. This is useful for:
  - software where source code is not available, or where software is a COTS application, or has proprietary dependencies
  - standardizing packaging for delivery
- Developers are responsible for using the correct base images, or for sourcing the correct libraries, runtimes, and middleware.
- Base image deprecation may not auto-propagate to deployed applications since the onus is on developers to update their images and re-deploy them.
- Operators cannot directly influence the choice of runtime versions (e.g. JDK version) and middleware (e.g. Tomcat version) that developers choose to use: these have to be enforced only via base images present in the private repositories.
- Apart from building individual images, developers generally have to do a lot of extra wiring up of their apps and services to create a complete, cohesive solution. Therefore, routine management operations like logging and service discovery can be time consuming to set up and operate.
  - For instance, in some cases, developers have to run an initial image as a container so that various Dockerfile instructions can be applied, and then commit the container and then run it again to test.
  - Sometimes, getting all the dependencies (ADD, COPY, VOLUME, etc.) and instruction order (RUN, CMD, ENTRYPOINT, ONBUILD, etc.) correct can be error prone and require some trial-and-error.
- When creating a systems topology, developers have to know a lot about the underlying system to correctly configure their inter-systems dependencies.
  - In the case of composable microservice applications, dependencies on other systems have to get put into the Dockerfile as ENV arguments; changes in where other APIs are hosted can cause images to be invalidated.
- In a polyglot language/runtime environment, platform operators still have to choose the languages/runtimes they will support, and this can limit choice for the developers if the base images for these are not made available, or kept patched by operators.

Buildpacks:

- With Buildpacks, developers only focus on their code and compiling/testing it via a CI/CD pipeline, they don't have to assemble the container.
- The CI/CD pipeline submits the compiled artifact to PCF where the pipeline can run various integration tests.
- Runtime versions and middleware versions are chosen by the platform operators, and the developers cannot override those choices.
- This also reduces the size of the artifacts that are created by developers. Instead of image payloads that are 100's of MB even for small applications, developers typically produce only small .jar and .war files (in the case of Java) or small .js files (in the case of Node). This enables developers to quickly push their payloads to the platform instead of having to commit large container images directly.
- With buildpacks, platform operators can provide support for a large number of languages and runtimes to their development community, since these are supported by Pivotal. New buildpacks can be built by organizations to suit their needs, and a large number of open source community buildpacks are also available from the community.
- Buildpacks are generally not suitable for COTS applications since these require specific configurations at the VM level, and are generally not designed to be cloud-native applications.

Image Scanning

PCF + Docker

In general, Docker image scanning systems have a "blacklist" approach: everything is OK except that which is in a CVE database.

- Using Docker's tools (or similar tools provided by any compatible 3rd party vendor), individual images are scanned in a private repository.
- For some base layer vulnerabilities, the entire image will have to be rebuilt, which implies that all developers have to incorporate those changes into their Dockerfiles.
- As the image repository grows, it takes longer and longer to run these scans.
- Since developers have significant flexibility in what goes in an image, the opportunities for vulnerabilities to slip into an image is higher.

PCF + Buildpacks

PCF, in general, takes a "whitelist" approach via buildpacks: only what is in the buildpack and host OS is ok. This gives a tighter compliance boundary. Additionally, PCF employs various techniques to ensure payload integrity, since instead of an image layer hash, it is looking at an application & library fingerprint. An example of PCF's added functionality for buildpack based deployments is:

- **Application Watchdog droplet scanning (coming in 2016, still under development):** Application Watchdog (AppDog) generates an authoritative "bill of materials" for each buildpack generated application on the platform. The bill of materials provides a list of dependencies found in the application droplet. This allows operators and developers to

meet compliance against versions, open source licensing (GPL, MIT, Apache, etc.), and vulnerabilities.

- This gives operators and developers a powerful tool to get fine grained assurance around their application package. AppDog looks to help answer questions like the following:
  - What applications contain or do not contain a specific dependency in their bill of materials? This can be useful to take newly found CVE reports and scan existing applications for the presence of these vulnerable dependencies.
  - What buildpack dependencies is an application actively using?
  - What version of a dependency is an application using?
  - What is the licensing model of the dependencies in use?

Image Patching

PCF + Docker

- Docker images are delivered to PCF pre-built, so PCF exerts no influence in how those images have been built. For Docker containers running on PCF, PCF cannot influence the layers in the image, and so containers have to be patched externally.
- An external container patching solution has to be implemented and kept updated with the latest CVE patches. This patching solution can be part of the standard CI/CD pipeline that builds the images within an organization.
- For VM host OS patching, PCF runs the same process as it does for buildpack applications: a new VM with the patched OS is stood up and all the containers from the respective old VM are recreated on the new VM: whether they're sourced from a droplet or a Docker image, the re-creation process is the same, and all containers always get to run on the always-patched VMs.

PCF + Buildpacks

PCF takes a hybrid approach to container + host OS patching.

- For buildpack apps, PCF creates the containers on the Cell VM itself. Therefore, the container is always based on what is available on the host VM.
- To patch a host VM, a new host VM is stood up with the latest OS updates. The payloads running on the corresponding old VM are re-created on the new VM. Since the containers are rebuilt with layers of the new host VM, the new containers are already patched (especially in the case of buildpack applications). The corresponding old VM is then destroyed after the routes to those apps are mapped to the new VM. This achieves a zero downtime host os update.
- For application level patching, since the runtimes and middleware are deployed via buildpacks, updating the buildpack will cause the new binaries to be used the next time the app is pushed.
  - Since apps are generally put into production via CI/CD pipelines, these pipelines can be triggered whenever there is an update to the buildpack to apply the latest changes to the deployed code.

Container Assurance

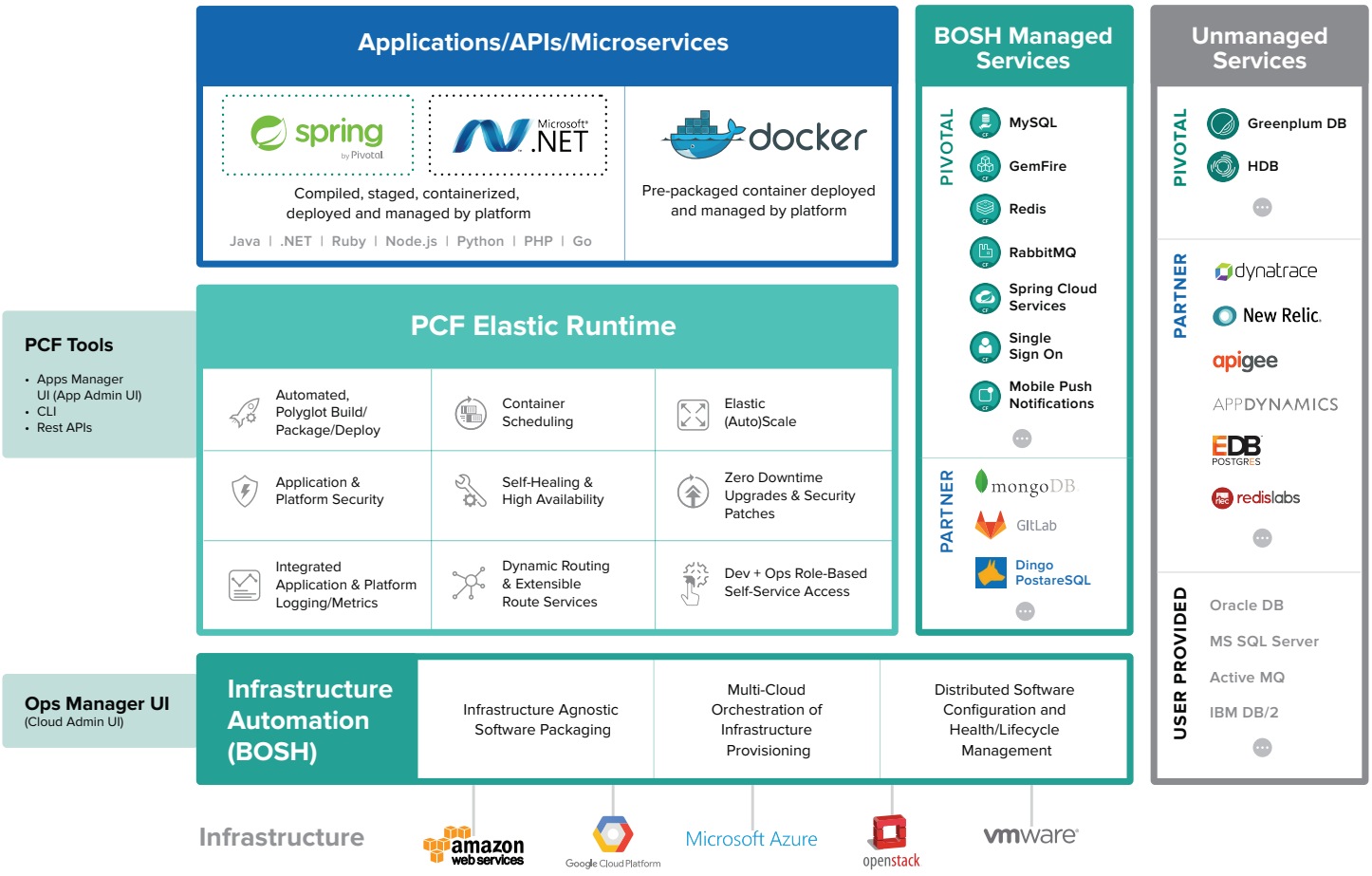
PCF treats containers as entirely disposable. PCF therefore provides various container integrity systems. For example, PCF discards a container when various triggering events occur, such as: when an application crashes, or when the host OS is updated, etc. Additionally, operators can entirely disable the feature of SSH'ing into containers, for example in production installations of PCF.

Therefore, whether a container was created from a buildpack or from a Docker image, all containers orchestrated by PCF get the same level of container assurance.

Appendix: Pivotal Cloud Foundry Overview

Pivotal Cloud Foundry has been built from the start to be an enterprise grade, application focused, container management and orchestration system. It design philosophy is to be an "app centric, cloud-native platform" that offers capabilities such as:

- A container-format-agnostic platform
- All payloads get the benefits of running on the platform:
  - High availability
  - Failover
  - Unified logging
  - Automatic scaling and traffic routing
  - Application Performance Management
- Rolling updates of host OS VMs without any downtime to the overall platform availability
- Blue-Green deployments for zero downtime application updates
- Application Security Groups for outbound traffic management
- Access to service brokers that allow name-based service binding to external services without recompiling code or rebuilding containers
- Multi-language support (Java, .NET, Node.js, Ruby, Go, PHP, etc.)
- Multi-cloud support (Vsphere, OpenStack, AWS, Microsoft Azure, Google Cloud Platform, etc.)
- Spring Framework (ex. Spring Boot, Spring Integration, Spring Cloud Netflix Services, Spring Cloud Dataflow, Spring Zipkin, Project Steeltoe, etc.)

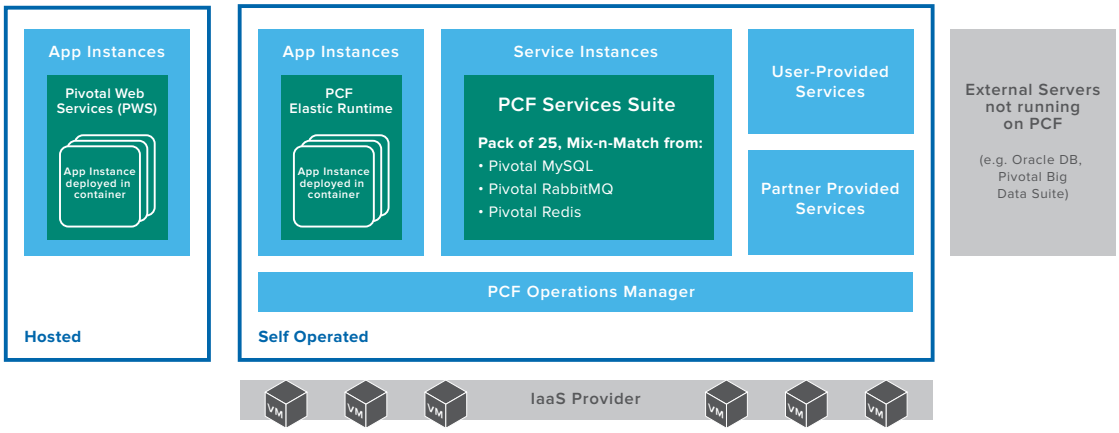


Pivotal

Out of the box, Pivotal provides a white labeled option for platform teams to customize with their corporate branding. This allows platform teams to offer PCF as an integrated offering along with their existing tooling and operational capabilities.

PCF is built for the enterprise. This includes recognizing that PCF has to fit into the existing ecosystem and workflows of an enterprise, while providing value and capability of its own. Often, enterprises already have some form of image ("Docker") creation framework: either as something under consideration, or as an internally developed toolchain. PCF provides incredible value and capability to these efforts from a container management and day-2 operations standpoint.

A high level, simplified diagram of PCF can be represented as follows:



The components shown above are described briefly here:

- **IaaS Provider (Infrastructure):** this is the virtualization layer that hosts a PCF instance. Various on-prem and off-prem options are supported by PCF, such as vSphere (usually used in an organization's on-prem data centers), and AWS/Microsoft Azure/Google Cloud Platform (usually used in an organization's public cloud account).
- **PCF Ops Manager and Elastic Runtime (Operations):** The Elastic Runtime is the set of tools that together form the full capability set of PCF (components like the cloud controller, Diego brain, Blobstore, Cell VM which run containerized payloads, etc. are part of this foundation). Platform operators manage this tier and developers are not involved in the maintenance and upkeep of this tier.
  - **Cells:** These are the VMs in the foundation that host the containers for applications deployed on the platform. The health and uptime of these VMs and the of the applications running in these Cells as containers is managed by the components in the foundation.
- **App Instances:** these are the actual payloads in the form of containers running in the various "Cells" indicated above. These containers are created from Buildpack droplets or from Docker images. These containers are scheduled, managed, and monitored by the components that make up the Elastic Runtime (e.g. Diego, Garden, etc.)

- **Service Instances:** just as PCF manages containerized processes in Cells (App Instances), it can manage middleware (such as databases, caches, etc.) running in managed VMs. PCF takes care of the health and uptime of these service VMs as well. Additionally, PCF doesn't restrict applications from using just the services that it is managing: applications can connect to any service they choose. PCF makes some services available as a convenience feature for developers and operators.
  - **Service Brokers:** are a powerful concept in PCF that provide a systematic means for applications to request access to services. The service broker is an API that advertises a catalog of services, and lets PCF instruct it to make its instances of those services available to applications in an automated manner. This allows developers to get access to middleware and other services without filing tickets or waiting for middleware to be provisioned by hand.
  - **User Defined Services:** applications can also be made to connect to arbitrary services that exist within the organization (or to which an organization has access to, such as a 3rd party SaaS service) and for which no service broker exists.

PCF and Spring Cloud Services

Pivotal is the custodian of the Spring Framework and has made significant investments in how developers can leverage the modern features of Spring (such as its formalizations of NetflixOSS components) within PCF with minimal effort.

Spring Cloud Services is a set of managed services within PCF that allow PCF to spin up microservice architectural components like Registry servers, Config servers, Circuit Breaker monitors, and Distributed Tracing monitors on demand, thus freeing developers from having to manage this "microservice connective tissue" and instead focus on the actual microservice components themselves.

Conclusion: Optimized for Development

PCF takes a structured, opinionated view of application deployment and management, and also of platform and application level "Day-2" ops. These terms specifically imply:

- **Structured:** a packaged, tested, solutioned, and fully supported set of components that are guaranteed to work together to achieve the overall tasks of the platform. This frees platform operators from having to stitch together a platform from a large and disparate ecosystem that offers components with overlapping or conflicting functionality, and each at varying states of maturity, operational readiness and commercial support.
- **Opinionated:** a set of overall methodologies, principles and best practices that underpin how the platform works, the capabilities it offers, and the dimensions of flexibility it provides. This frees platform operators and developers from "reinventing the wheel" regarding routine development, deployment, and operational tasks, and lets them focus on business centric tasks such as optimized microservices architectures, capacity management, etc.

An Application-Centric View

In the end, it is Pivotal's and Pivotal Cloud Foundry's view that business value does not simply lie in things like container formats, image construction, or payload orchestration. These are all means to an end, not ends unto themselves.

Business value resides in developer code and the applications that are a consequence of that code. Business value is realized when application code is put into production in a fast, simple, and repeatable process (**Development Velocity**), and when that production code runs in a resilient, fault tolerant, highly maintainable, automated platform (**Operational Excellence**).

PCF aims to solve for business value by abstracting away all the complexity of the actual implementation level details for achieving Development Velocity and Operational Excellence, and therefore helps businesses evolve into software driven enterprises.