Pivotal Cloud Foundry (PCF) is a multi-cloud platform for the deployment, management, and continuous delivery of applications, containers, and functions. PCF is a distribution of the open source Cloud Foundry developed and maintained by Pivotal Software, Inc. PCF is aimed at enterprise users and offers additional features and services—from Pivotal and from other third parties—for installing and operating Cloud Foundry as well as to expand its capabilities and make it easier to use. Major cloud platforms such as Amazon Web Services and Google Cloud also provide templates and quickstarts that automate large portions of the PCF deployment process.

Pivotal Cloud Foundry abstracts away the process of setting up and managing an application runtime environment so that developers can focus solely on their applications and associated data. Running a single command—`cf push`—will create a scalable environment for your application in seconds, which might otherwise take hours to spin up manually. PCF allows developers to deploy and deliver software quickly, without needing to manage the underlying infrastructure.

In this post, we'll explore each of the technologies that make up a typical Pivotal Cloud Foundry cluster and how they work together. If you're already familiar with PCF architecture, feel free to continue on to part two to dive right into PCF's key performance metrics.

Getting abstract

Pivotal Cloud Foundry creates two kinds of virtual machines that handle different aspects of an elastic runtime environment:

- component VMs create the underlying infrastructure for a deployment
- host VMs provide a generic runtime environment for applications

Component VMs allow PCF to be cloud-agnostic by providing a standardized infrastructure environment for staging and running applications on any supported cloud hosting service—including AWS, Google, Azure, OpenStack, and vSphere. The generic runtime provided by host VMs enables a single PCF deployment to host multiple applications regardless of language or dependencies. Once an application is pushed via `cf push`, PCF stages and packages it into a binary droplet that can be distributed and executed on a host VM.

In addition to abstraction, PCF provides significant scaling capabilities. Global-scale companies like Boeing, Wells Fargo, and Citi have adopted and used PCF to migrate and build applications in an elastic, cloud-based environment. A deployment can run any number of applications as long as there are sufficient underlying infrastructure resources. Hundreds of thousands of application instances can be spread across over a thousand host VMs. To increase availability and redundancy, PCF provides a single command to scale up the number of application instances and, if your environment is deployed across multiple availability zones, PCF will automatically spread instances across zones. Component VMs can likewise be scaled up and assigned to multiple zones to ensure high availability.

Monitoring Pivotal Cloud Foundry

Much of the information in this guide is targeted toward PCF operators, which is the term Pivotal applies to people managing and monitoring a PCF deployment (as opposed to developers or end users, who deploy or access applications running on the deployment). For operators, monitoring your deployment's performance and capacity indicators is vital to ensuring that applications are running optimally. It's also necessary to check that the deployment can scale to match demand, whether scaling horizontally in terms of the number of application instances or vertically in terms of the resources available within the VMs. We'll cover the key metrics operators will want to collect and monitor in PCF.

Additionally, Pivotal Cloud Foundry offers several ways to monitor application performance. In parts three and four we will also look at how developers who are deploying to PCF can collect monitoring data from their applications.

Before diving into monitoring a PCF deployment, it's important to understand its pieces and how they work.

Key components of Pivotal Cloud Foundry architecture

PCF is a distributed system comprising many components that run, manage, and monitor the health of the deployment and its application runtime environment, which might be hosting dozens or hundreds of applications. The primary components we will cover in this guide, along with their subsystems, are:

- BOSH/the Ops Manager (deployment automation)
- The User Account and Authentication server (identity management)
- The Gorouter (application and system routing)
- The Cloud Controller (application staging and running)
- Diego (application execution and runtime)
- Loggregator (logs and metric aggregation)

BOSH and the Ops Manager

BOSH is a deployment manager that can automatically provision and deploy widely distributed, cloud-based software. Originally developed specifically for Cloud Foundry, BOSH can also be used outside of Cloud Foundry environments, for example to deploy a ZooKeeper or Kubernetes cluster.

Essentially, BOSH is what allows PCF to be deployed in any cloud by providing an interface to build required infrastructure components on top of a given IaaS platform. BOSH handles the deployment of the underlying PCF infrastructure by launching and managing all required component VMs via the **BOSH Director**.

Through common plugins such as the Health Monitor, BOSH can track the health of its VMs and also self-heal if it detects that a VM has crashed or has otherwise become inaccessible. It will attempt to recreate the faulty VM automatically to avoid downtime.

The BOSH Director reads YAML deployment manifests to determine what VMs, persistent disks, and other resources are required, as well as how many availability zones to use. For a given cloud provider, the BOSH Director relies on an IaaS-specific manifest, or cloud config, that is applied on top of a baseline deployment config manifest for PCF. The cloud config maps PCF resources to specific resources for a given cloud provider, for instance mapping PCF availability zone `z1` to AWS availability zone `us-east-1a` or Google Cloud zone `us-central1-f`.

The Director launches VMs built on **stemcells**, which include a base operating system, a BOSH agent for monitoring, and any required utilities and configuration files. **Releases**, which are layered on top of the stemcell, contain a versioned set of configurations, source code, binaries, scripts, and anything else that might be needed to run a specific software package on the VM.

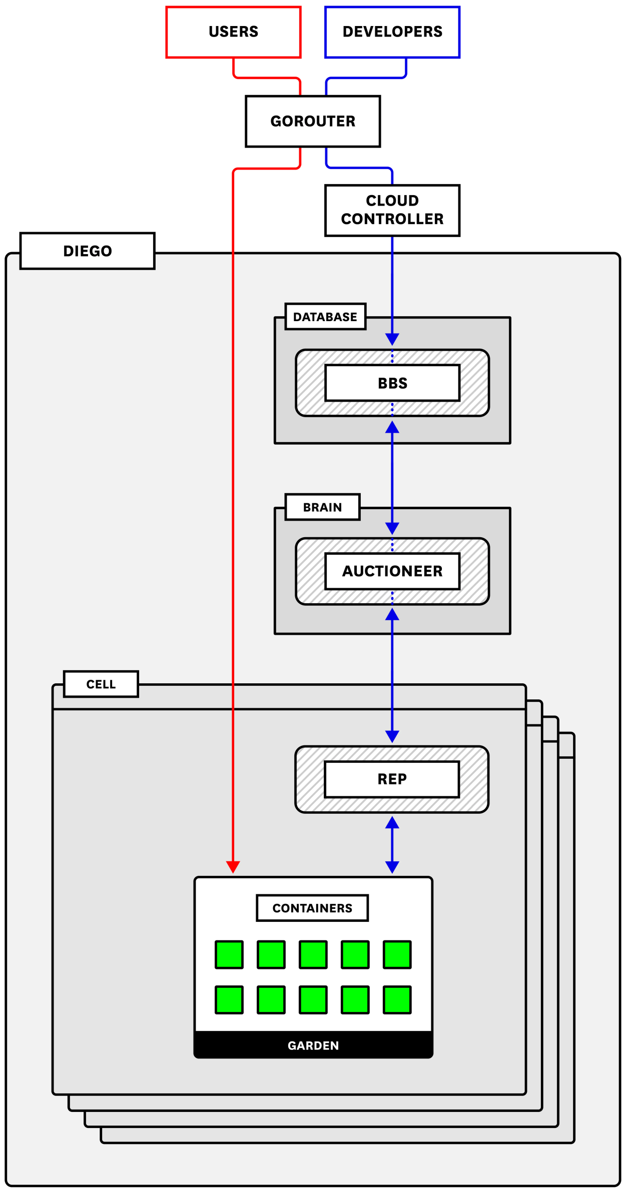
Ops Manager

PCF wraps the BOSH API with the **Ops Manager**, which provides a web-based GUI to automate tasks and help administer the deployment. The Ops Manager generates manifest files from the options an operator has selected and sends them to the BOSH Director. The Ops Manager also provides various extensions and services for PCF that are packaged as tiles, which an operator imports and configures before deploying to the cluster. These services include monitoring solutions, several of which we will discuss in detail in part threeof this series.

Within a PCF deployment, operators can also use the Ops Manager to install their desired application runtime environments (Linux or Windows) on host VMs. BOSH releases are available as buildpacks, which can be installed to provide runtime support for applications and services.

User Account and Authentication

The User Account and Authentication (UAA) server is Cloud Foundry's central identity and access management (IAM) component. PCF operators can use the UAA API to create and manage user accounts. UAA also acts as an OAuth2 server, and can generate authentication tokens for client applications based on the access scope they've been granted.


USERS
DEVELOPERS
GOROUTER
CLOUD CONTROLLER
DIEGO
DATABASE
BBS
BRAIN
AUCTIONEER
CELL
REP
CONTAINERS
GARDEN

A high-level overview of Pivotal Cloud Foundry architecture

Gorouter

The **Gorouter** is PCF’s router (written in Go). It handles and routes incoming requests. These requests come either from operators or developers sending commands—such as `cf push`—that are then routed to the Cloud Controller API, or from end users accessing application instances running on the deployment. The Gorouter communicates with the Diego BBS (explained below) to keep track of which applications exist, how many instances there are, and where they are running in order to maintain a routing table and shuttle user traffic appropriately. It provides basic round-robin load balancing for sending traffic to available application instances.

The Gorouter only supports requests over HTTP/HTTPS. If you use TLS encryption and want it to terminate as close to the instance as possible (as opposed to terminating at a load balancer) you may optionally enable and configure TCP routing. In this case, a separate TCP router handles routing for TCP traffic.

Cloud Controller

Pivotal Cloud Foundry’s **Cloud Controller (CC)** provides API endpoints for operators and developers to interact with PCF and the applications running on a specific domain. Commands include staging applications, starting or stopping applications, collecting health information, and querying that all desired applications are running. The CC is also responsible for communicating with the User Account and Authentication server to authenticate the user and ensure that they have the proper permissions to perform the requested task.

Developers can stage applications that are buildpack based or that are Docker images. Buildpack-based applications require one or more buildpacks to provide dependencies. For example, the Python buildpack is needed to stage a Django application. Docker images don’t require buildpacks because the image contains everything required to run the application. The staging process differs somewhat between these two types of applications. We will cover both options below.

Buildpack applications

When a developer pushes an application for staging, the CC sends needed files and instructions to PCF's container orchestration and management component, called **Diego**. For staging and running applications, the CC requires a MySQL database and a blobstore.

The CC creates a record and stores application metadata in the database including the application name; applicable orgs, spaces, services, and user roles; the number of instances to spin up; memory and disk quotas; etc. It also will create and bind a route to the new application.

The CC packages and stores required binary files—generated at various points in the staging process—in the blobstore. These binaries consist of five types: application packages (source code, resource files, etc.), buildpacks (e.g., application dependencies), a resource cache (larger files from the application package), a buildpack cache (larger files created during application staging), and droplets (the fully staged and ready-to-run application).

The CC sends instructions to Diego to distribute and execute the staging tasks, which take everything the application needs and package it into a droplet that can be run on a Cloud Foundry container. When complete, Diego sends the staged application droplet for storage in the CC's blobstore and notifies the CC that the application is ready. Finally, after staging, the Cloud Controller signals Diego to start the application and continues to communicate with Diego for updates on the application's status.

Once an application is running, the CC makes it possible to bind services to the application. Services provision reserved resources for an application on demand. They can provide a wide range of types of resources. A few examples might be a web application account, a set of environment variables, or a dedicated Redis cluster.

Docker image applications

The main difference with how the CC stages Docker image–based applications is that buildpacks are not applied because the image itself provides the dependences. The CC sends the image to Diego for staging, receives and stores required metadata about the image from Diego, and then instructs Diego to schedule processes to run the application.

Diego

Diego is the container orchestration system for Cloud Foundry deployments, having replaced the previous DEA (Droplet Execution Agent). Diego handles the creation and management of the containers that stage and run applications. PCF operators can use the Ops Manager to choose which runtime backend they want to use—the Guardian backend for Linux or Garden Windows for Windows (or both).

An application's lifecycle, described above, on Diego is largely the same in either case. These backends are managed through the **Garden API**. When a developer deploys their application with `cf push`, Diego uses Garden to create a generic, abstracted, containerized environment for any kind of instance, whether it be a buildpack-based droplet, a Docker image, or a Windows Server container.

Diego includes a health monitor and is self-healing: it will attempt to restart instances that have crashed in order to ensure that the number of running instances matches what the deployment configuration requires.

To accomplish its orchestration tasks, Diego relies on three main components:

- the Diego Brain
- the database VM
- one or more Diego cells

Before discussing these Diego components, it's important to understand **tasks** and **Long-Running Processes (LRPs)**, as these concepts are fundamental to how Cloud Foundry runs applications.

Tasks and LRPs

Pivotal Cloud Foundry translates incoming application-specific requests and processes into generic, abstracted tasks or Long-Running Processes (LRPs). This abstraction lets PCF forward all requests and processes to Diego so that it can schedule and assign them to host VMs known as cells for execution.

Tasks are one-off processes—that is, tasks are inherently terminating and provide a success/failure response. A task could be a database migration, or an initial service setup (for example, staging tasks for getting an application up and running). Containers running tasks are destroyed once the task is complete.

LRPs are one or more application instances or other processes that are generally continuous, scalable, and always available. PCF attempts to ensure high availability and resilience by maintaining multiple running instances of the same LRP across availability zones.

Diego and the Cloud Controller work together to ensure that the number of running LRPs (`ActualLRPs`) matches the number that the system should be running (`DesiredLRPs`). That is, they make sure that Diego is always running as many application instances as users are expecting and will automatically terminate or spin up new instances if there are discrepancies.

Diego Brain

The main purpose of the Diego Brain is to schedule and assign incoming requests to the cells for execution. The primary component responsible for this is the **Auctioneer**. The Auctioneer receives work from the Cloud Controller via the BBS (discussed below). It then communicates with a cell via the cell's **Rep** (the point of contact between the cell and the rest of the deployment) to auction off this work after the **Stager** translates it into Tasks and LRPs. This process lets PCF balance load and maintain high availability as much as possible. The Auctioneer attempts to assign tasks and LRPs in batches. Its auction algorithm operates with the following descending priorities: ensure that all LRPs have at least one instance running at all times; assign all tasks/LRPs in the current batch; distribute process load across running instances and availability zones to ensure availability.

The Diego Brain's other components carry out additional functions, including maintaining correct LRP counts, storing and handling resources required for application staging, or providing SSH access to application containers.

Database VM

Diego's database VM is essentially responsible for monitoring, storing, and updating the state of the deployment and of the work that is assigned to the cells. There are two primary components within the database VM:

- the bulletin board system (BBS)
- Locket

The **BBS** is the intermediary between the CC and the Auctioneer and provides an API to communicate with and send requests to the Diego cells, which in turn create the containers that run the requested work. As such, it is the gateway through which information about `DesiredLRPs` and `ActualLRPs` flows and is vital to maintaining an accurate picture of the Diego cluster.

The BBS requires its own relational database (MySQL or Postgres) to maintain a record of cell status, unallocated work, and other information. It uses this database to keep an up-to-date image of all the work the Diego cluster is handling and sends that to the Auctioneer when it is assigning a new batch so that work can be distributed appropriately. This also helps avoid duplicating LRPs or Tasks.

The BBS also runs regular convergence assessments that compare the running state on the Diego cells against the desired state provided by the Cloud Controller to ensure they are the same.

Locket uses a key-value store and provides an API for service discovery and registering locks. Certain components must register locks for processes to ensure that, for example, there are no conflicts resulting from multiple cells accepting the same work.

Diego cells

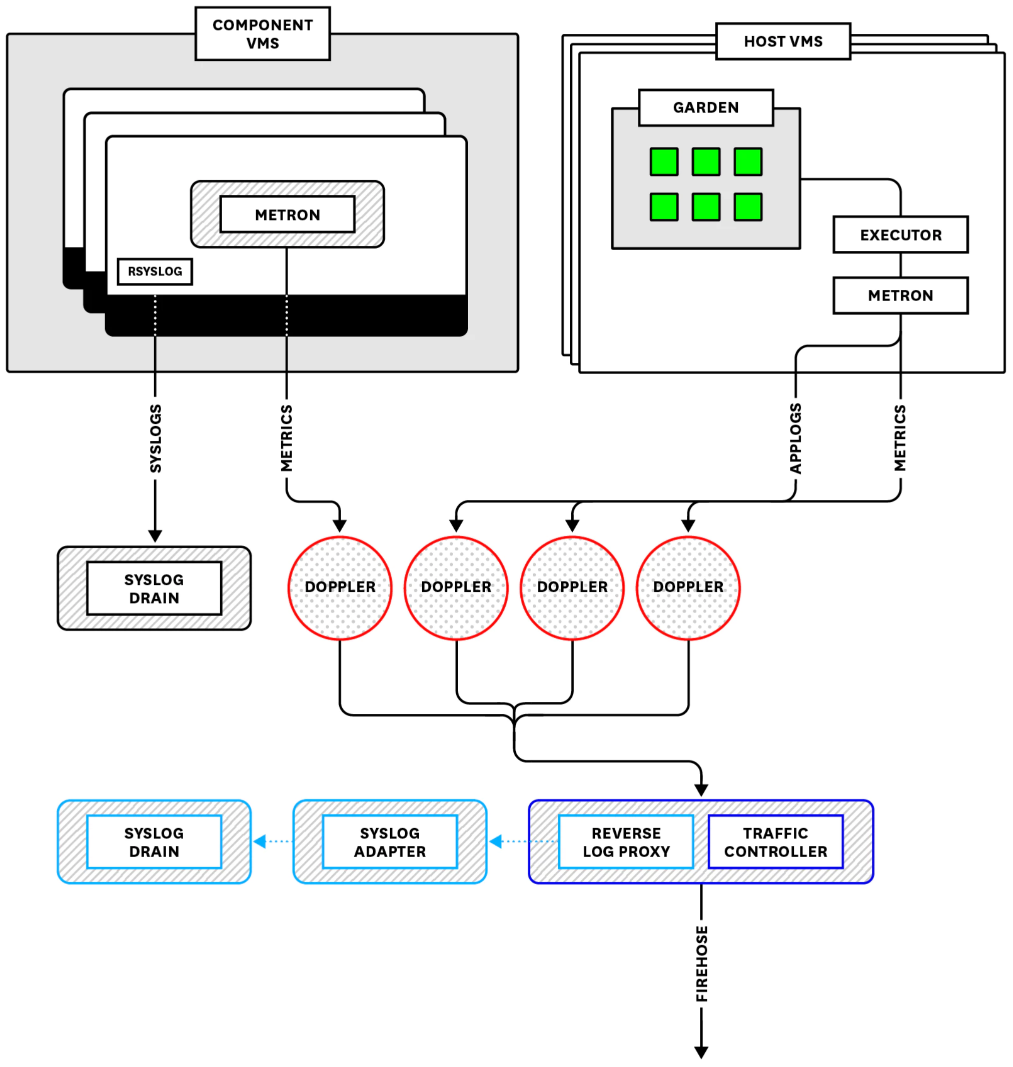
Diego cells are host VMs that run the containers doing the actual work. Multiple Diego cells can be running in the deployment using different stemcells and releases, or even Garden backends. The Rep on each cell performs a number of functions: for starters, it registers and maintains the cell's presence with the BBS, and it communicates with the Auctioneer to bid for auctioned jobs. When a task or LRP is accepted, the Rep's internal process, the **Executor**, instructs Garden to create a container to run it. The Rep also forwards information about completed tasks to the BBS.

The Rep constantly monitors the containers running within its cell to make sure they are healthy and compares their number and state to what the BBS expects to confirm that there are no discrepancies. The Rep also monitors resource allocation in comparison to the cell's capacity. Finally, it forwards metrics and logs from the containers to the central Loggregator system (described below).

Another important component in the Diego cells is the **Route-Emitter**. This registers and records changes to LRP states and emits updated routing tables to the Gorouter to make sure that application instances are accessible.

Loggregator

Loggregator is a system that aggregates and streams logs and (despite its name) metrics from PCF infrastructure components as well as instrumented applications running on the deployment. Logs and metrics in PCF come in a few different flavors. Loggregator collects platform metrics from PCF components, system-level resource metrics from the VMs, and application logs. Loggregator will also collect available metrics from installed services. That is, some PCF add-on products (for example, Redis) publish their own metrics, which are then forwarded to Loggregator and included in a data stream called the Firehose.



An overview of how PCF aggregates and streams application logs, application and component metrics, and system logs.

Application logs include any logs that an application writes to `stderr` or `stdout`. They also include log messages emitted by PCF components as they process requests related to staging, running, and interacting with an application. Note, however, that Loggregator does not include PCF component *system logs*, or logs produced by internal processes and written to files on the VMs. Instead, these are streamed through rsyslog and can be accessed by connecting a third-party syslog drain.

Loggregator translates, or “marshalls,” messages into envelopes based on the log or metric type. This processes uses the dropsonde protocol, which abstracts and standardizes metric and event metadata to be processed downstream. For application logs and metrics coming from Diego cells, this process is handled by the Diego Executor, which reads off of the containers’ `stdout` and `stderr` outputs. Other PCF components forward metrics via the StatsD protocol to a StatsD-injector that translates them into dropsonde. These translated logs and metrics are then sent to processes that run on the VMs called **Metron Agents**, which forward them using gRPC to **Doppler servers**.

Dopplers separate and package envelopes coming in from the Metrons into protocol buffers, sometimes called sinks, based on the envelope type. As of version 2 of the Loggregator API, there are five types of envelopes: `log`, `counter`, `gauge`, `timer`, or `event`. (See the Loggregator API documentation for more information.)

Dopplers hold messages temporarily before sending them on to one of two destinations. By default, they go to a **Traffic Controller**, which aggregates everything from all availability zones before shuttling it on in a stream that is accessible either via the CF logging API (accessed by `cf logs`) or via the Firehose (covered below).

The Dopplers can also send logs to a **Reverse Log Proxy** (using gRPC), which is colocated on the Traffic Controller VM and forwards them to a **Syslog Adapter** that transforms the messages into standard syslog format. They then can be accessed by one or more third-party syslog drains that users can bind to an application. Binding a drain is accomplished by deploying a syslog drain release.

By default, Dopplers are unable to store logs.* If logs coming from a specific buffer cannot be consumed as quickly as they are being produced or forwarded, information will simply be wiped to make room for the next batch of logs. This means that scaling each component properly is important. Having an appropriate number of Metrons, Dopplers, and Traffic Controllers will help ensure no logs are dropped on the floor. We will look more at how to scale these components in part two.

* As of PCF 2.2, Log Cache, a process colocated on the Dopplers, can store application log messages for on-demand access via a RESTful interface. We will discuss Log Cache further in part three.

BOSH metrics

BOSH includes an agent on each VM it provisions that produces heartbeats every minute, which include system-level resource metrics for that VM. The agent will also emit alerts regarding lifecycle events. As of PCF version 2.0, BOSH system metrics are included in the Loggregator stream by default. A BOSH System Metrics Plugin reads these heartbeats and alerts and sends them to the BOSH System Metrics Server, which in turn streams them via gRPC to the BOSH System Metrics Forwarder, colocated on the Traffic Controller VM. These metrics and events then join the Loggregator stream.

Firehose

The **Firehose** is a WebSocket endpoint that clients can connect to in order to access the full Loggregator stream. The data streaming from the Firehose can be accessed using **nozzles**, components that can consume information from the Loggregator stream and decode the messages from the protocol buffers. Access to the Firehose requires a user with special permissions, given the possibly sensitive nature of information included in the logs.

The Firehose is where operators can get the bulk of the information available about their Cloud Foundry deployment and its health. We will discuss ways of accessing the Firehose stream in more detail in part three of this series.

Diving into PCF metrics

In this post, we’ve explored the main components that make up a Pivotal Cloud Foundry deployment. In the next post, we will cover some of the key performance metrics and indicators that can help PCF operators monitor these components, as well as the overall health of their cluster.

Acknowledgments

We wish to thank Amber Alston, Katrina Bakas, Matt Cholick, Jared Ruckle, and the rest of the Pivotal Cloud Foundry team for their technical review and feedback for this series.