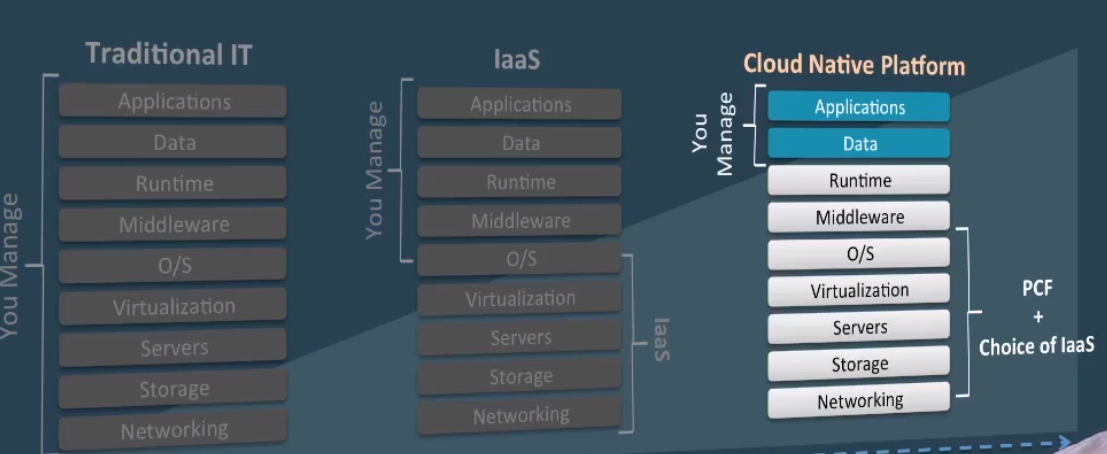
**Pivotal Cloud Foundry Developer**

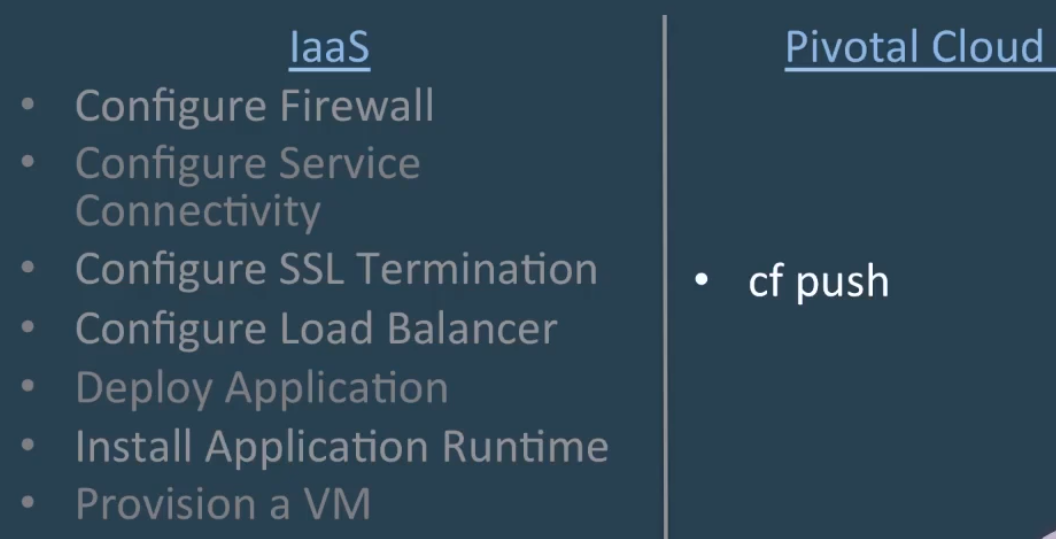
Application PAAS

1. **Evolution of Cloud:**

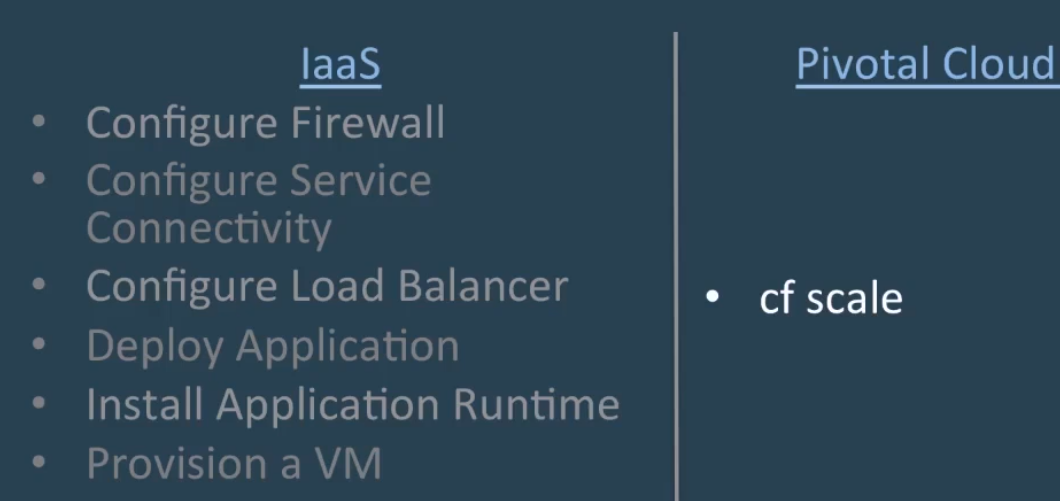


1. **Difference between IAAS and PCF**

*Deploying an application*



*Scaling an application*



**PCF Core Tenets:**

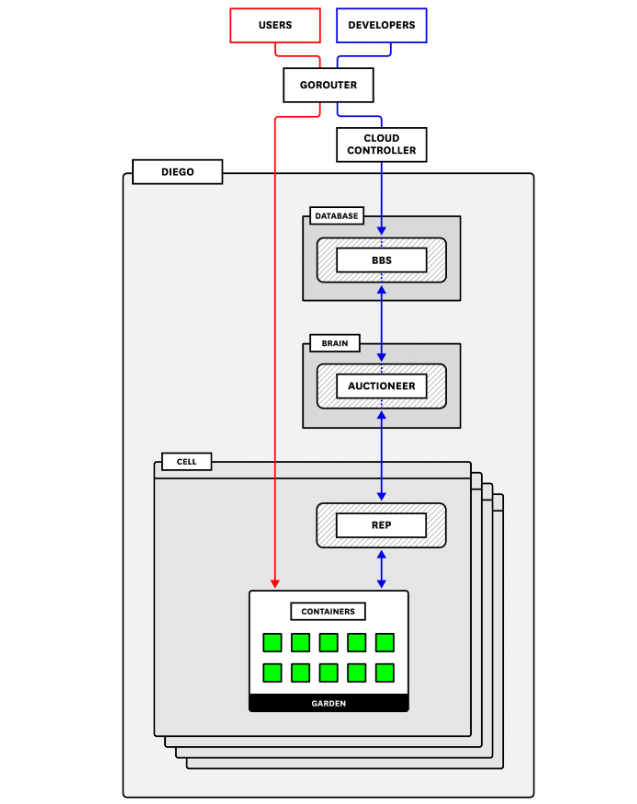
Self serve platform for Developers

IAAS Independent

CF Push for deploying apps in PCF

**12 Factor App:**

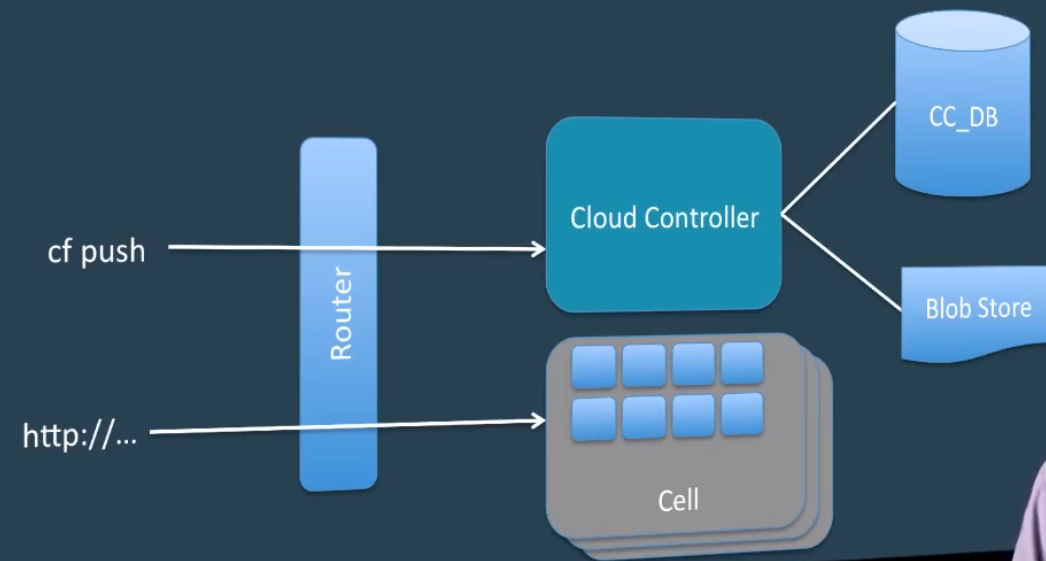
1. Execute the apps as one or more stateless processes
2. Scale out via the process model
3. Maximize robustness with fast start up and graceful shut down
4. Treat logs as event streams
5. One Codebase tracked in revision control, many deploys
6. Explicitly declare and isolate dependencies
7. Store Config in the environment
8. Treat backing services as attached resources
9. Strictly separate build and run stages
10. Export services via port binding
11. Keep Development, staging and production as similar as possible
12. Run admin/management tasks as one-off processes
13. **Elastic Runtime Architecture**

****

1. *GORouter(Application and System Routing)*

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.



1. *Cloud Controller API(Application staging and Running)*

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.

CC Bridge translates app specific messages into generic language of tasks and LRPs

1. *Diego(Application Execution and Runtime) : Schedules tasks and long running processes*

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

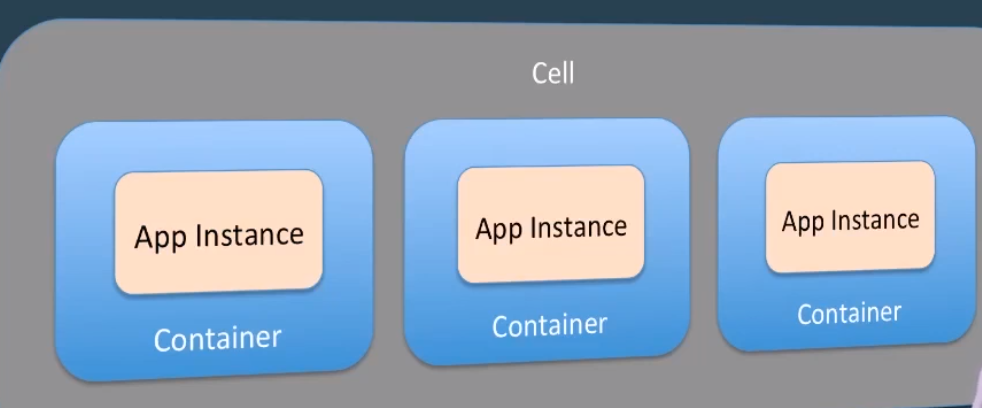
CELL:

Task : It is guaranteed to run atmost once eg : stage an application

Long Running Process: It’s a web app. There can be multiple instances

Container : Tasks and Long Running process are run within an container

Cell : containers are within Cell



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** (API to access the Diego database for tasks and long running processes). 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. This is done with the help of Converger.

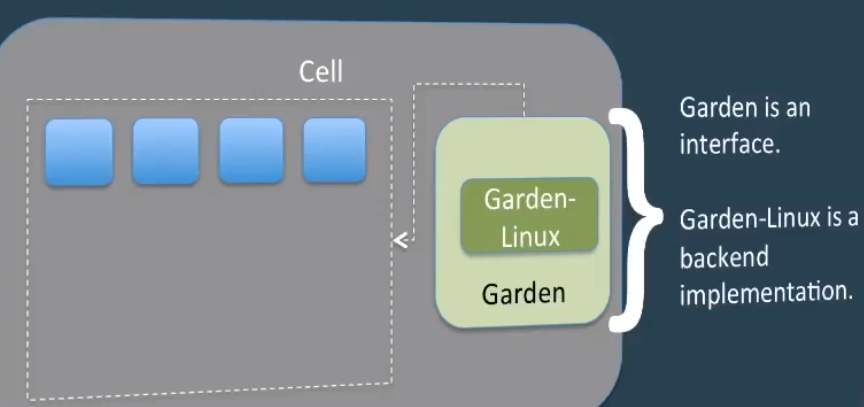
**Locket** uses a key-value store and provides an API for service discovery and registering locks. Certain components must register locks for processes to ensure that, for example, there are no conflicts resulting from multiple cells accepting the same work.

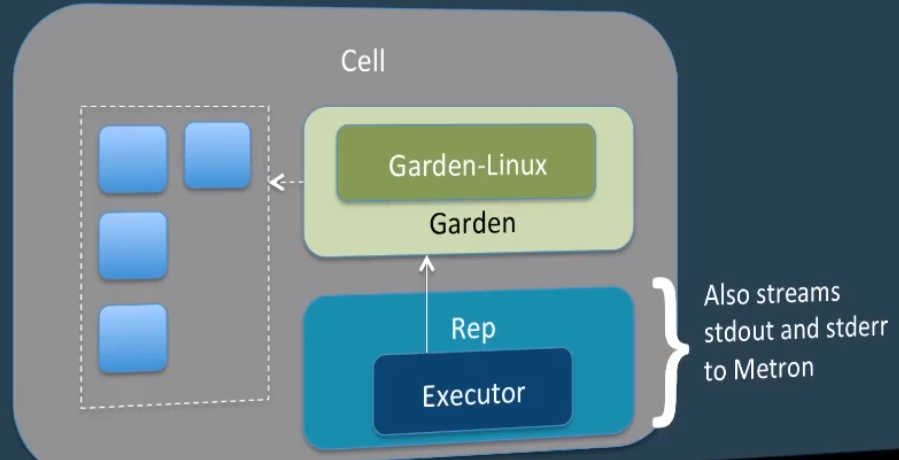
DIEGO CELLS:

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(**Sub process of the Rep which manages container allocations on the cell. It logs on to the metron which sits on the cell and which can forwards logs to loggregator systems**)**, instructs **Garden**(who manages all cells) to create a container to run it. The Rep also forwards information about completed tasks to the BBS.

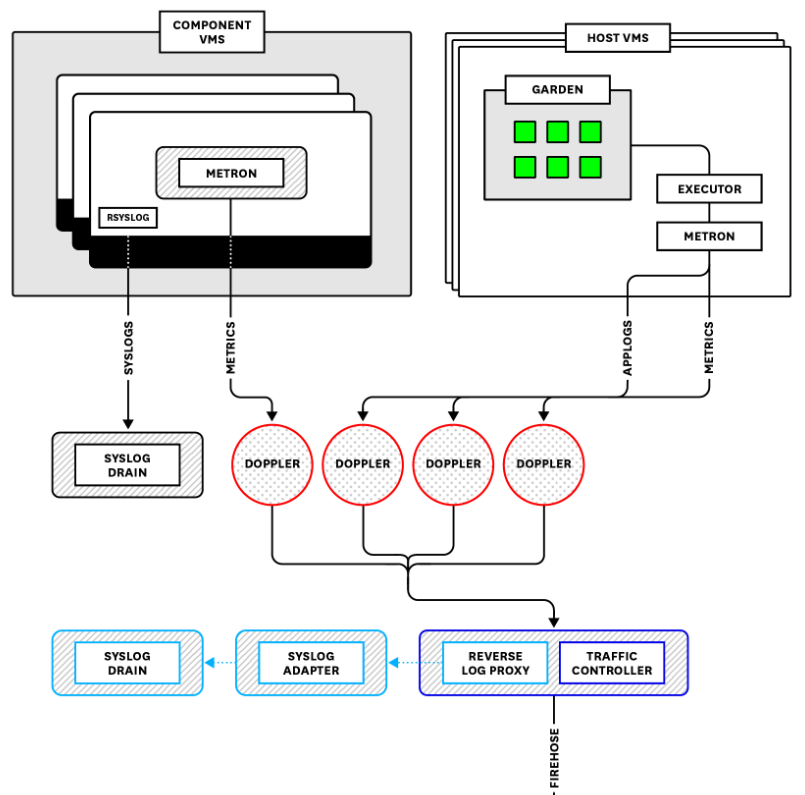
The Rep constantly monitors the containers running within its cell to make sure they are healthy and compares their number and state to what the BBS expects to confirm that there are no discrepancies. The Rep also monitors resource allocation in comparison to the cell’s capacity. Finally, it forwards 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.





1. *Loggregator(*Gathers logs from Metron which sits on the cell*)*



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.

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**(Doppler is one of the subsystems in loggregator which creates app syslog drains which drains the logs to third party tool such as Splunk or papertrail

).

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.

Dopplers hold messages temporarily before sending them on to one of two destinations. By default, they go to a **Traffic Controller**(Traffic Controller is one of the subsystems in loggregator which handles client requests for logs and also exposes web socket endpoint called firehose), 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.

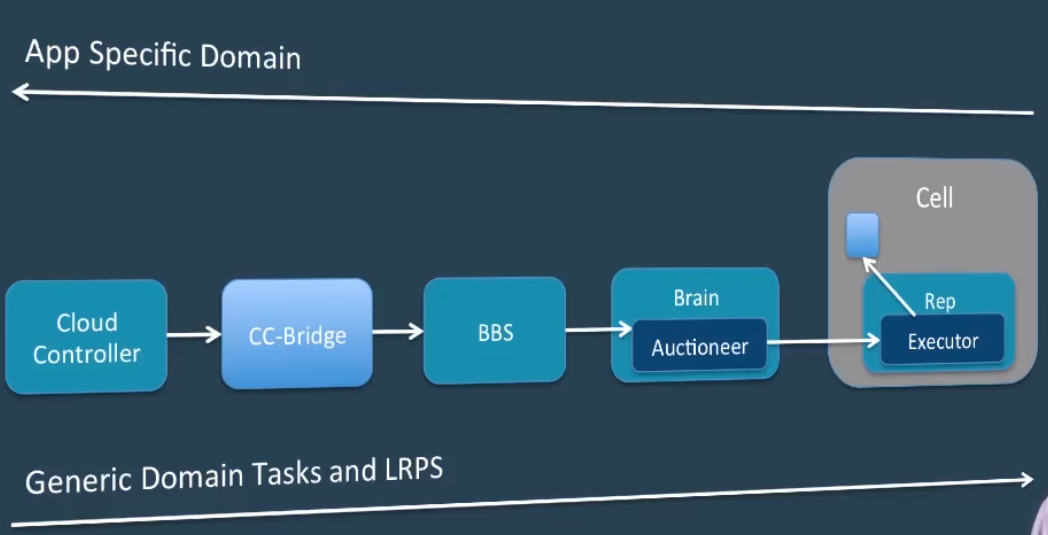
By default, Dopplers are unable to store logs.\* If logs coming from a specific buffer cannot be consumed as quickly as they are being produced or forwarded, information will simply be wiped to make room for the next batch of logs. This means that scaling each component properly is important. Having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

\* 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.

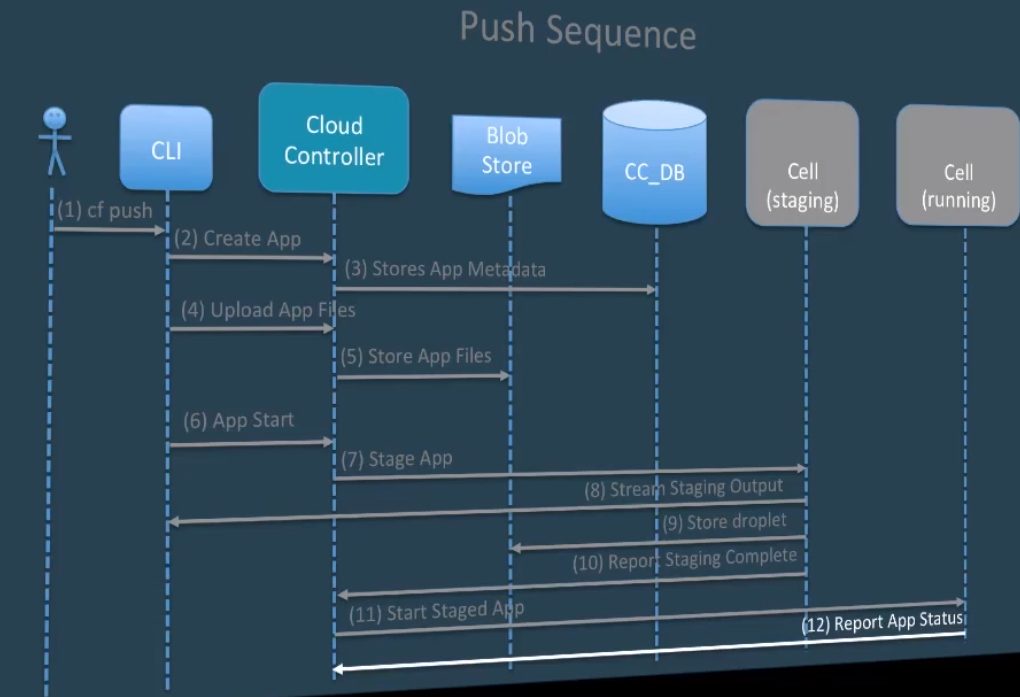
**Firehose**: A web socket endpoint that exposes app logs, container metrics and ER Component metrics. Does not include ER Component logs

**Nozzles** : Consumes firehouse output

**Stage and Run Request Flow**

****

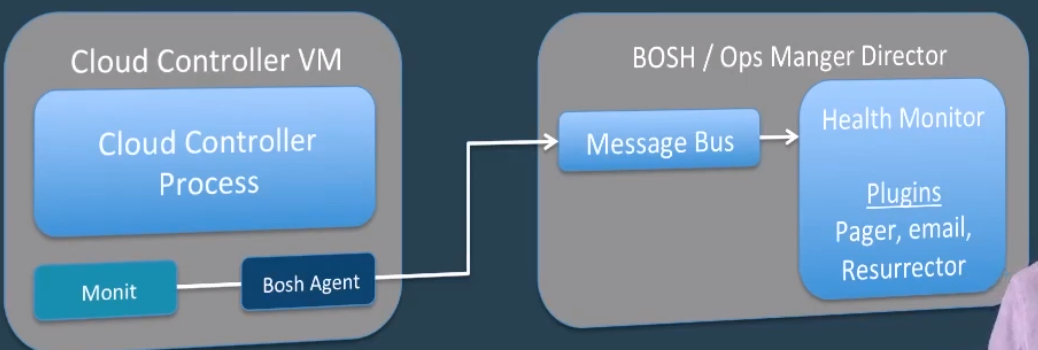
CC pass the request to CC Bridge. Then it gets captures in BBS. Then it submits the request in Auctioneer. Then the winning cell will provision the same for the application

**

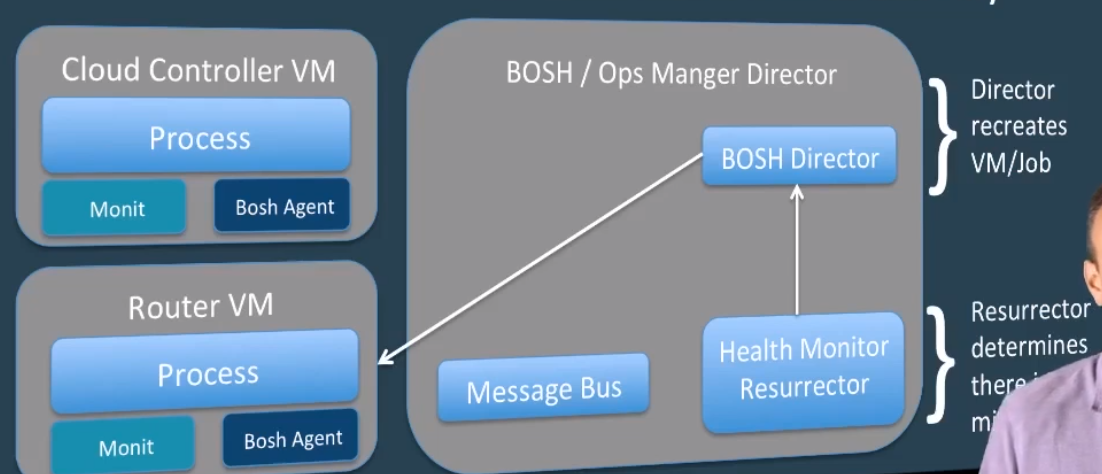
1. **High Availability:**

Application instances are evenly distributed across availability zones. In the event,we loose the zone, app will be up and running.

Bosh agent : ER processes are monitored and automatically restarted. Monit in Bosh agent will take care of this. Restart event is reported back to the Health monitor for further investigation.



Resurrector : what if the entire VM gets destroyed. In normal scenario, bosh agent will send heart beat to health monitor in appropriate interval. If the entire VM is down, resurrector will understand there is a missing job as there are no messages coming from that particular VM’s bosh agent and Bosh Director will recreate the VM.



Scaling : when the number of app instance is scaled, the app will not be stopped, just the additional containers will be created. But when the memory is scaled, the app will be stopped for sometime. This is because container will be dropped and the new container will be created with the requested memory.

Cf scale <appname> -i 2

Events : Basically to see the history of events that has occurred with the app.

Levels of HA:

Self Healing(Brain within Diego),

Logging,

Troubleshooting with events,

Scaling(Vertical(increasing memory of container) and Horizontal(increasing instances))

1. **Services:**

All services will be stored in environment variable VCAP\_SERVICES

Binding an existing services to your application.

* Managed Services – These are advertised in the marketplace.
* User provided Services – These are not provisioned outside the platform. We can connect two apps together here

1. **Manifest:**

All application manifest information can be stored in the .ymil file. You can make changes to this file and the changes will be reflected when the cf push command is issued the next time

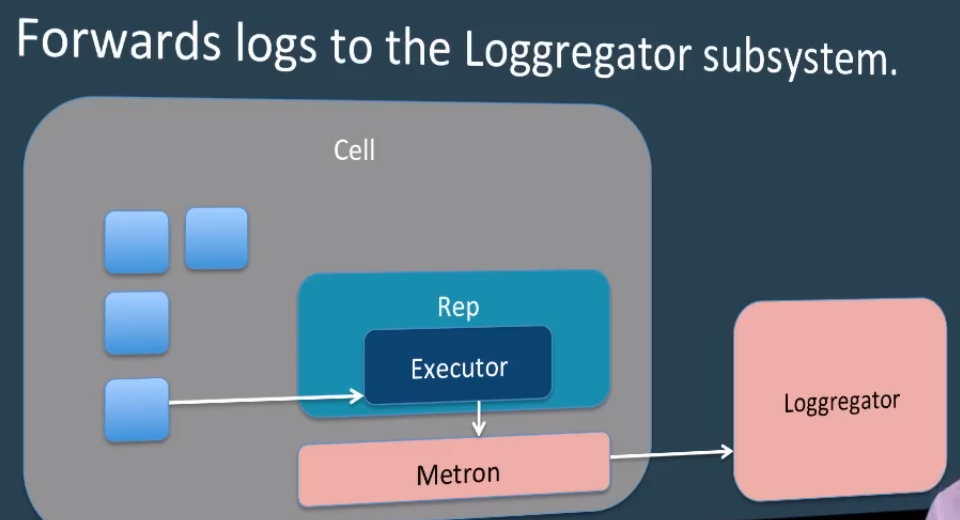
1. **Application Security Groups:**

These are virtual firewall that will control the outbound traffic from the application. Security groups get designed through Security Group Rules file.

Cf security-groups

Cf security-group <group name>

1. **Log Drain**

****

It is basically created to persistent the logs in a third party tool like papertrial/Splunk so that it can be used at any later point of time

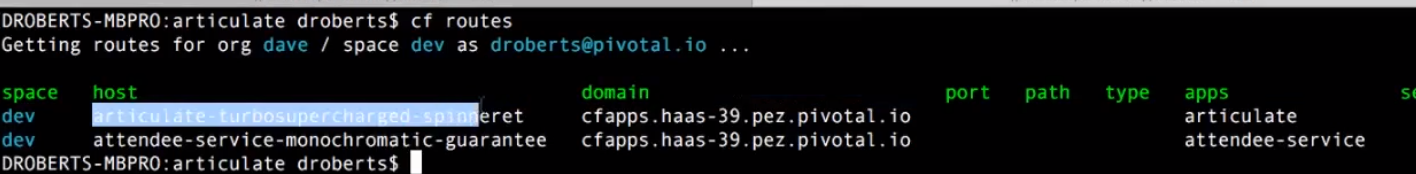
**References :** <https://www.datadoghq.com/blog/pivotal-cloud-foundry-architecture/>

1. **Blue Green Deployment and Microservices**

Blue Green Deployment is basically used to achieve zero downtime

Steps :

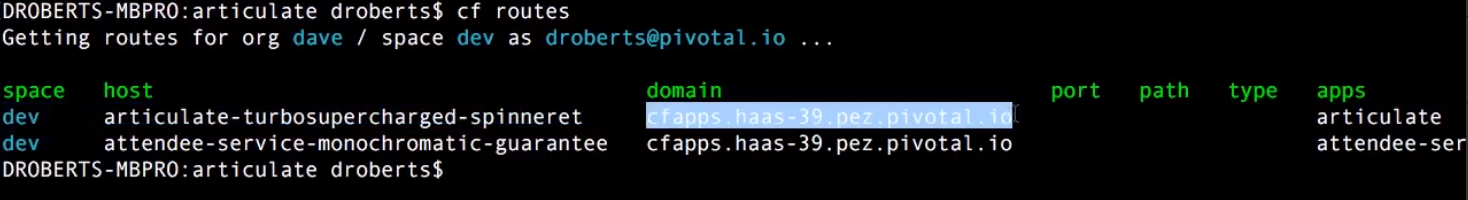
1. Have the new version ready
2. Get host of the current production application using cf routes and have that ready



1. Now we are going to push the newer version using

*cf push <application-new-version> -p <jarname> -m <memory allocation> -n<current production hostname url copied from the previous step and suffix it with temp> --no -start*

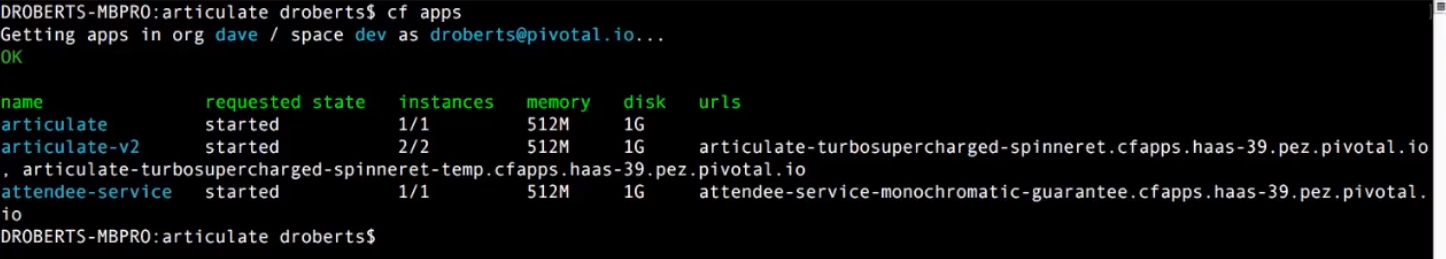
1. Then start the new version using cf start <new version>
2. Now test the new application version by hitting the respective url
3. Now we have to map the production route(get production domain from cf apps) to the new application version.



*Cf map-route <new application version> <Current production domain> -n <current production host>*

1. Now we have to unmap the route from the current production version

*cf unmap-route <current production version> <Current production domain> -n <current production host>*



1. Then unmap the temporary route from new application production version.

*cf unmap-route <new application version> <Current production domain> -n <temp host name with the suffix temp>*



1. **Application Auto scaler, Performance monitor and Metrics**

Cf m – to view the market place

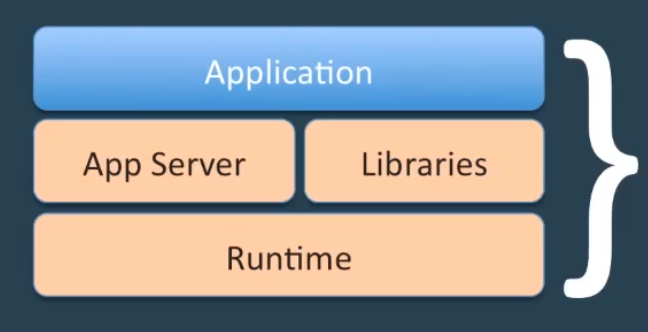
Cf m -s <service name> - To know more about the service

Cf create-service <service name> gold autoscaler – To create an auto scalar.

Cf bind-service <app name> <service name> - To bind the service to the application

**k. Build Pack and Service Broker**

Droplet(tarball)



*BuildPack API*

bin/detect – Determines whether the buildpack can stage the application

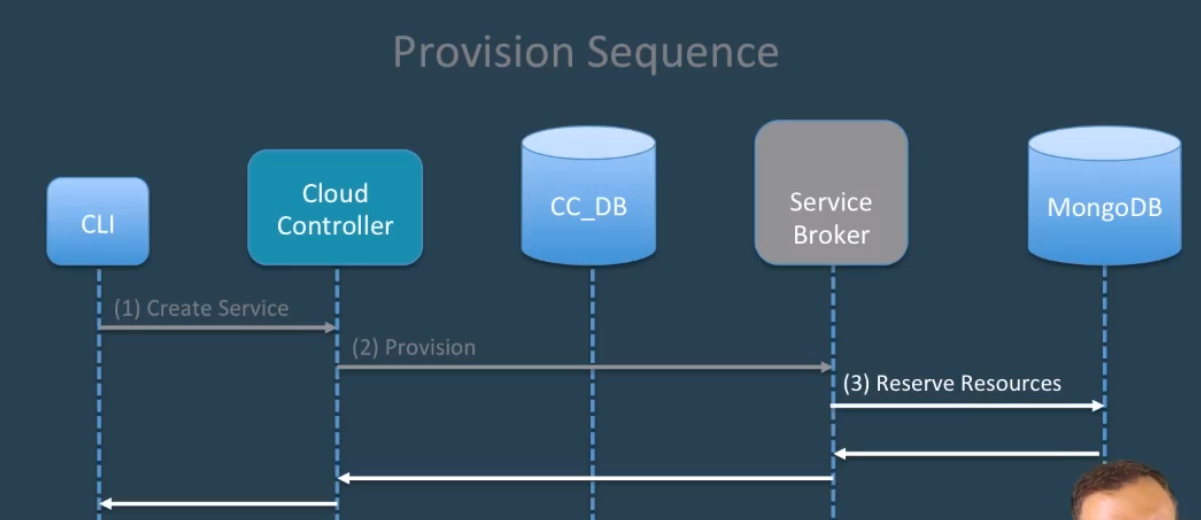
bin/compile – Builds the droplet

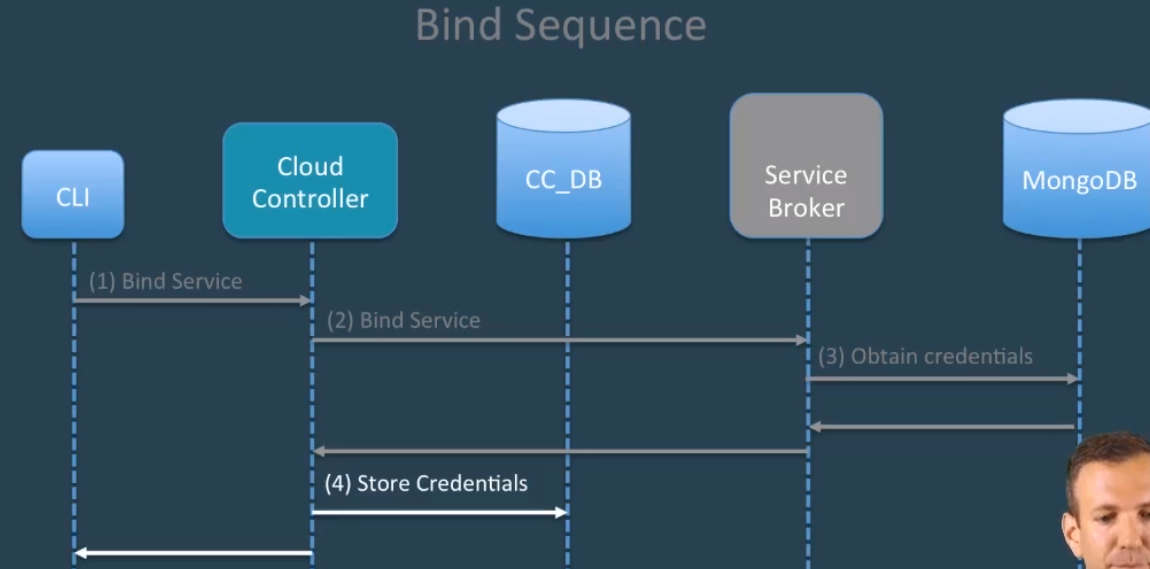
bin/release – Provides information on how to run the application

Cf buildpacks – To get the list of available build packs

*Service Broker API*

Managed Services should implement the Service Broker API(REST Based API over http).

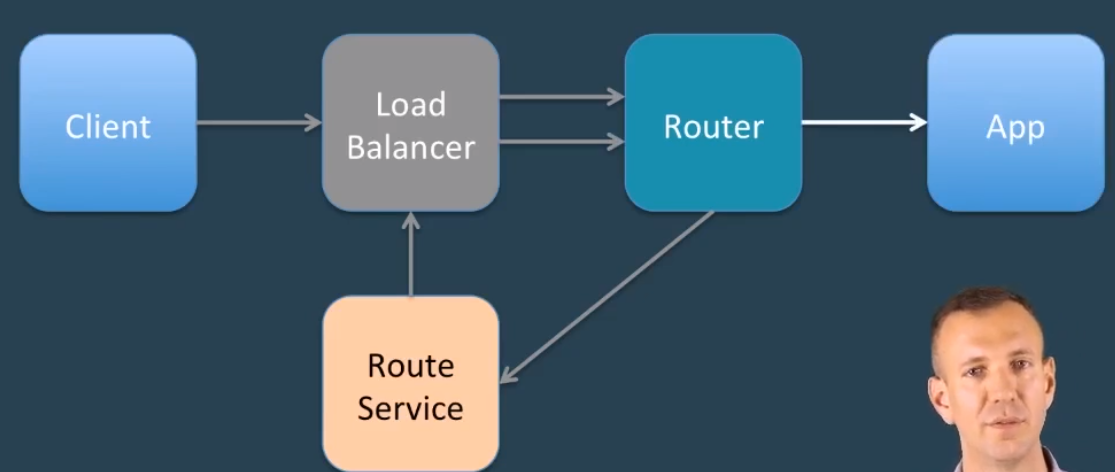




Cf create-service-broker <ServiceBrokerName> <UserName> <Password> <Url> --Space -scoped

**l. Continuous Delivery and Route Service**

**Route Service Flow**

****

Request from the client goes to the Load Balancer. From there, it goes to the router. Router then passes the request to Route Service along with 3 headers(X-CF-Forwarded-Url[URL of the application route], X-CF-Proxy-Signature, X-CF-Poxy-Metadata[The last 2 headers are used by the Router to validate the request and pass through to the application ])