O'REILLY®

Compliments of dynatrace

Cloud Foundry
The Definitive Guide

DEVELOP, DEPLOY, AND SCALE

FREE CHAPTERS

Duncan C. E. Winn

Cloud Foundry: The Definitive Guide

This excerpt contains Chapters 1–3, 6, and 9 of the book *Cloud Foundry: The Definitive Guide*. The complete book is available at *www.safaribooksonline.com* and through other retailers.

Develop, Deploy, and Scale

Duncan C. E. Winn

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Cloud Foundry: The Definitive Guide
by Duncan C. E. Winn

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Anderson and Virginia Wilson
Production Editor: Melanie Yarbrough
Copyeditor: Octal Publishing, Inc.
Proofreader: Christina Edwards
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2017: First Edition

Revision History for the First Edition
2017-05-22: First Release

978-1-491-93243-8

[LSI]

Table of Contents

CHAPTER 1

The Cloud-Native Platform

Cloud Foundry is a platform for running applications, tasks, and services. Its purpose is to change the way applications, tasks, and services are deployed and run by significantly reducing the *develop-to-deployment* cycle time.

As a cloud-native platform, Cloud Foundry directly uses cloud-based infrastructure so that applications running on the platform can be infrastructure unaware. Cloud Foundry provides a contract between itself and your cloud-native apps to run them predictably and reliably, even in the face of unreliable infrastructure.

If you need a brief summary of the benefits of the Cloud Foundry platform, this chapter is for you. Otherwise, feel free to jump ahead to Chapter 2.

Why You Need a Cloud-Native Platform

To understand the business reasons for using Cloud Foundry, I suggest that you begin by reading *Cloud Foundry: The Cloud-Native Platform*, which discusses the value of Cloud Foundry and explores its overall purpose from a business perspective.

Cloud Foundry is an "opinionated" (more on this later in the chapter), structured platform that imposes a strict contract between the following:

- The infrastructure layer underpinning it
- The applications and services it supports

Cloud-native platforms do far more than provide developers self-service resources through abstracting infrastructure. Chapter 2 discusses at length their inbuilt features, such as resiliency, log aggregation, user management, and security. Figure 1-1 shows a progression from traditional infrastructure to Infrastructure as a Service (IaaS) and on to cloud-native platforms. Through each phase of evolution, the value

line rises due to increased abstraction. Your responsibility and requirement to configure various pieces of the software, middleware, and infrastructure stack in support of your application code diminish as the value line rises. The key is that cloud-native platforms are designed to do more for you so that you can focus on delivering applications with business value.

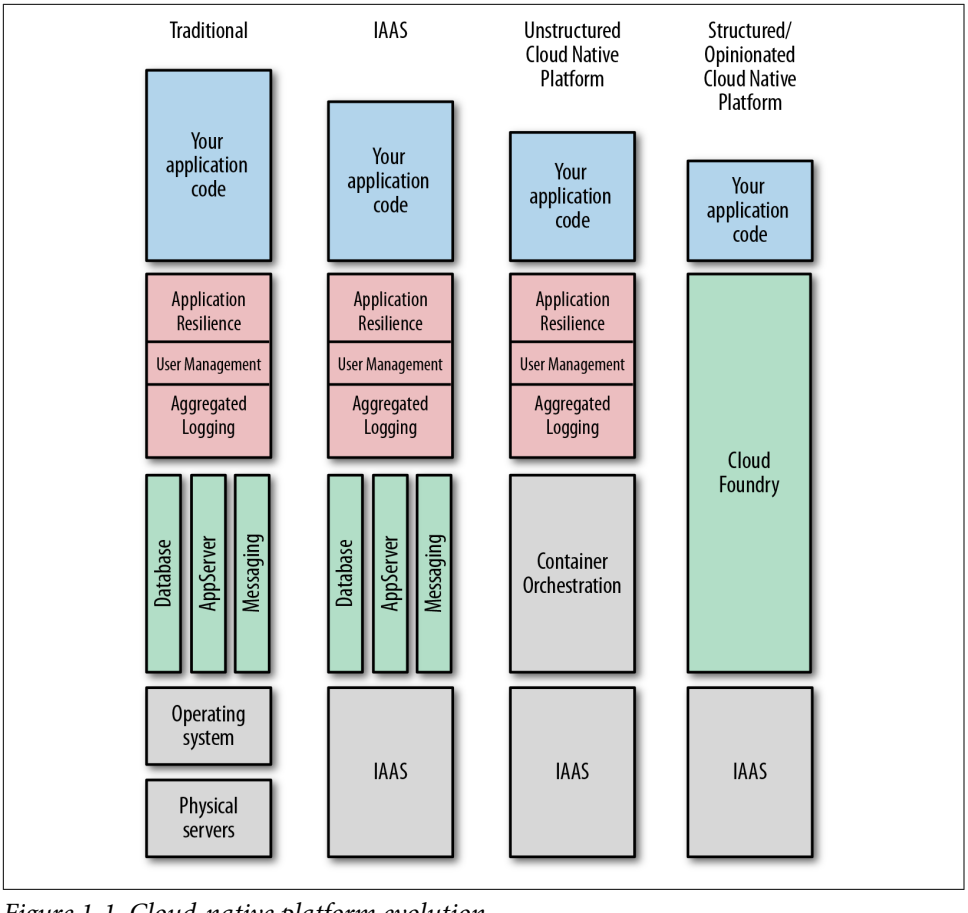

Figure 1-1. Cloud-native platform evolution

Cloud-Native Platform Concepts

In the Preface, I pointed out that Cloud Foundry's focus is not so much what a platform is or what it does, but rather what it enables you to achieve. It has the potential to make the software build, test, deploy, and scale cycle significantly faster. It removes many of the hurdles involved in deploying software, making it possible for you to release software at will.

Specifically, here's what the Cloud Foundry platform offers:

Services as a higher level of abstraction above infrastructure
: Cloud Foundry provides a self-service mechanism for the on-demand deployment of applications bound to an array of provisioned middleware and routing services. This benefit removes the management overhead of both the middleware and infrastructure layer from the developer, significantly reducing the development-to-deployment time.

Containers
: Cloud Foundry runs all deployed applications in containers. You can deploy applications as container images or as standalone apps containerized by Cloud Foundry. This provides flexibility. Companies already established with Docker can deploy existing Docker images to Cloud Foundry. However, containerizing applications on the user's behalf offers additional productivity and operational benefits because the resulting container image is built from known and vetted platform components. This approach allows you to run your vulnerability scans against your trusted artifacts once per update. From this point, only the application source code requires additional vulnerability scanning on a per deployment basis. Essentially, there is less to check on a per deployment basis because all of your supporting artifacts have already been vetted.

Agile and automation
: You can use Cloud Foundry as part of a CI/CD pipeline to provision environments and services on demand as the application moves through the pipeline to a production-ready state. This helps satisfy the key Agile requirement of getting code into the hands of end users when required.

Cultural shift to DevOps
: Cross-cutting concerns is a well-understood concept by developers. Adopting Cloud Foundry is ideally accompanied by a cultural shift to DevOps, meaning that you need to break down traditional walls, team silos, and ticket-based handoffs to get the most benefit from it.

Microservices support
: Cloud Foundry supports microservices through providing mechanisms for integrating and coordinating loosely coupled services. To realize the benefits of microservices, a platform is required to provide additional supporting capabilities; for example, Cloud Foundry provides applications with capabilities such as built-in resilience, application authentication, and aggregated logging.

Cloud-native application support
: Cloud Foundry provides a contract against which applications can be developed. This contract makes doing the right thing simple and will result in better application performance, management, and resilience.

Not all cloud-native platforms are the same. Some are self-built and pieced together from various components; others are black-boxed and completely proprietary. The Cloud Foundry cloud-native platform has three defining characteristics: it is structured, opinionated, and open. I'll examine each of these traits in the following sections.

The Structured Platform

Within the platform space, two distinct architectural patterns have emerged: structured and unstructured:

- Structured platforms provide built-in capabilities and integration points for key concerns such as enterprise-wide user management, security, and compliance. With these kinds of platforms, everything you need to run your applications should be provided in a repeatable way, regardless of what infrastructure you run on. Cloud Foundry is a perfect example of a structured platform.
- Unstructured platforms have the flexibility to define a bespoke solution at a granular level. An example of an unstructured platform would involve a *"build your own platform"* approach with a mix of cloud-provided services and home-grown tools, assembled for an individual company.

Structured platforms focus on simplifying the overall operational model. Rather than integrating, operating, and maintaining numerous individual components, the platform operator just deals with the one platform. Structured platforms remove all the undifferentiated heavy lifting: tasks that must be done—for example, service discovery or application placement—but that are not directly related to revenue-generating software.

Although structured platforms are often used for building new cloud-native applications, they also support legacy application integration where it makes sense to do so, allowing for a broader mix of workloads than traditionally anticipated. The structured approach provides a much faster "getting started" experience with lower overall effort required to operate and maintain the environment.

The Opinionated Platform

When you look at successful software, the greatest and most widely adopted technologies are incredibly opinionated. What this means is that they are built on, and adhere to, a set of well-defined principles employing best practices. They are proven to work in a practical way and reflect how things can and should be done when not constrained by the baggage of technical debt. Opinions produce contracts to ensure applications are constrained to do the right thing.

Platforms are opinionated because they make specific assumptions and optimizations to remove complexity and pain from the user. Opinionated platforms are designed to be consistent across environments, with every feature working as designed out of the box. For example, the Cloud Foundry platform provides the same user experience when deployed over different IaaS layers and the same developer experience regardless of the application language. Opinionated platforms such as Cloud Foundry can still be configurable and extended, but not to the extent that the nature of the platform changes. Platforms should have opinions on how your software is deployed, run, and scaled, but not where an application is deployed; this means that, with respect to infrastructure choice, applications should run anywhere.

The Open Platform

Cloud Foundry is an open platform. It is open on three axes:

- It allows a choice of IaaS layer to underpin it (Google Cloud Platform [GCP], Amazon Web Services [AWS], Microsoft Azure, VMware vSphere, OpenStack, etc.).
- It allows for a number of different developer frameworks, polyglot languages, and application services (Ruby, Go, Spring, etc.).
- It is open-sourced under an Apache 2 license and governed by a multi-organization foundation.

Closed platforms can be proprietary and often focus on a specific problem. They might support only a single infrastructure, language, or use case. Open platforms offer choice where it matters.

Summary

Cloud Foundry is an opinionated, structured, and open platform. As such, it is:

- built on, and adheres to, a set of well-defined principles employing best practices.
- constrained to do the right thing for your application, based on defined contracts.
- consistent across different infrastructure/cloud environments.
- configurable and extendable, but not to the degree that the nature of the platform changes.

For the developer, Cloud Foundry provides a fast "on rails" development and deployment experience. For the operator, it reduces operational effort through providing built-in capabilities and integration points for key enterprise concerns such as user management, security, and self-healing.

Now you understand the nature of Cloud Foundry. Chapter 2 focuses on explaining Cloud Foundry's underlying concepts.

CHAPTER 2
Concepts

This chapter explains the core concepts underpinning Cloud Foundry. Understanding these concepts paints a complete picture of why and how you should use the platform. These concepts include the need to *deal with undifferentiated heavy lifting* and why *cloud-based operating systems* are essential in protecting your cloud investment. This chapter also touches on the philosophical perspectives behind Cloud Foundry with its opinionated *do more* approach. Operational aspects, including release engineering through BOSH, and built-in resilience and fault tolerance are also introduced. Finally, some of the core capabilities of the platform beyond container orchestration are introduced, including the aggregated streaming of logs and metrics and the user access and authentication (UAA) management.

Undifferentiated Heavy Lifting

Cloud Foundry is a platform for running applications and one-off tasks. The essence of Cloud Foundry is to provide companies with the speed, simplicity, and control they need to develop and deploy applications. It achieves this by undertaking many of the burdensome boilerplate responsibilities associated with delivering software. These types of responsibilities are referred to as undifferentiated heavy lifting, tasks that must be done—for example, container orchestration or application placement—but that are not directly related to the development of revenue-generating software. The following are some examples of undifferentiated heavy lifting:

- Provisioning VMs, OSs, middleware, and databases
- Application runtime configuration and memory tuning
- User management and SSO integration
- Load balancing and traffic routing

- Centralized log aggregation
- Scaling
- Security auditing
- Providing fault tolerance and resilience
- Service discovery
- Application placement and container creation and orchestration
- Blue/green deployments with the use of canaries

If you do not have a platform to abstract the underlying infrastructure and provide the aforementioned capabilities, this additional burden of responsibility remains yours. If you are spending significant time and effort building bespoke environments for shipping software, refocusing investment back into your core business will provide a huge payoff. Cloud Foundry allows enterprises to refocus effort back into the business by removing as much of the undifferentiated heavy lifting as possible.

The Cloud Operating System

As an application platform, Cloud Foundry is infrastructure-agnostic, sitting on top of your infrastructure of choice. As depicted in Figure 2-1, Cloud Foundry is effectively a cloud-based operating system that utilizes cloud-based resources, which are hidden and abstracted away from the end user. As discussed in Chapter 1, in the same way that the OS on your phone, tablet, or laptop abstracts the underlying physical compute resource, Cloud Foundry abstracts the infrastructure's compute resource (specifically virtual storage, networking, RAM, and CPU). The net effect is that Cloud Foundry serves both as a standard and efficient way to deploy applications and services across different cloud-computing environments. Conversely, if you are stuck with directly using IaaS–specific APIs, it requires knowledge of the developer patterns and operations specific to the underlying IaaS technology, frequently resulting in applications becoming tightly coupled to the underlying infrastructure.

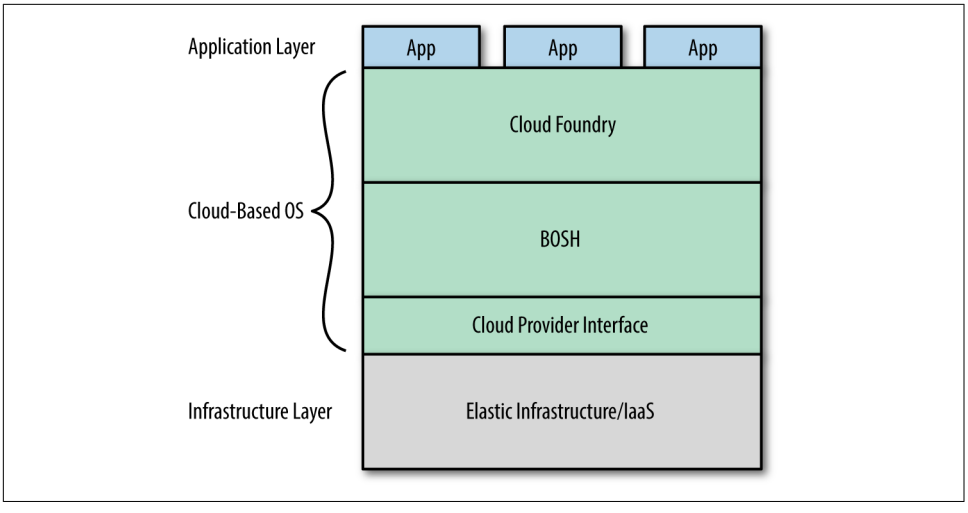

Figure 2-1. Cloud Foundry layers forming a cloud-based OS

Do More

Historically the long pole of application delivery, the part on the critical path that blocks progress, has been the IT department. This results in a concept I call server hugging, whereby developers hold on to (and hug) a plethora of VMs just in case they need them again someday.

Nowadays, businesses no longer need to be constrained by lengthy IT processes or organizational silos. Cloud Foundry provides a contractual promise to allow businesses to move with velocity and establish a developer–feedback loop so that they can tightly align products to user expectations. With Cloud Foundry, product managers *get their business back* and IT engineers can focus on more interesting issues and *get to eat dinner at home*.

Platforms are concerned not only with providing environments and middleware for running applications. For example, Cloud Foundry takes on the responsibility of keeping applications up and running in the face of failures within the system. It also provides security, user administration, workload scheduling, and monitoring capabilities. Onsi Fakhouri, Pivotal's Vice President of Research and Development, famously tweeted this haiku:

> Here is my source code,
> run it on the cloud for me.
> I do not care how!

Onsi's quote captures the essence of Cloud Foundry's *do more* capability. Cloud Foundry is about doing more on behalf of both the developer and operator so that they can focus on what really differentiates the business. This characteristic is seen all

throughout the Cloud Foundry ecosystem. You can take a similar approach with BOSH, Cloud Foundry's release-engineering system, and state, *"Here are my servers, make them a Cloud Foundry. I do not care how!"*

The Application as the Unit of Deployment

Traditionally, deploying application code required provisioning and deploying VMs, OSs, and middleware to create a development environment for the application to run in. After that environment was provisioned, it required patching and ongoing maintenance. New environments were then created as the application moved through the deployment pipeline.

Early incarnations of platforms centered on middleware: defining complex topology diagrams of application servers, databases, and messaging engines into which you could drop your application. When this topology diagram (or blueprint) was defined, you then specified some additional configuration such as IP addresses and ports to bring the defined topology and applications into existence. Although this was a step in the right direction, from a developer's perspective there was still a layer of complexity that you needed to configure for each deployment.

Cloud Foundry differs from traditional provisioning and orchestration engines in a fundamental way: it uses middleware and infrastructure directly, allowing streamlined development through self-service environments. Developers can build, deploy, run, and scale applications on Cloud Foundry without having to be mindful of the specific underlying infrastructure, middleware, and container implementation.

Cloud Foundry allows the *unit of deployment*, i.e., what you deploy to run your application, to be isolated to just the application itself. Even though there are some benefits to encapsulating both your application and dependencies as a precomposed container image, I still believe it is more secure and more efficient to keep just the application as the unit of deployment and allow the platform to handle the remaining concerns. The trade-offs between both approaches are discussed further in Chapter 5; however, the benefit of Cloud Foundry is that it supports both approaches. Buildpacks are discussed at length in that chapter, but for now, it is enough to know that buildpacks provide the framework and runtime support for your applications. A specific buildpack is used to package your application with all of its dependencies. The resulting *staged* application is referred to as a *droplet*.

On-boarding developers is easy; they can deploy applications to Cloud Foundry using existing tool chains with little to no code modification. It enables the developer to remove the cost and complexity of configuring infrastructure for their applications. Using a self-service model, developers can deploy and scale applications without being directly locked into the IaaS layer.

Because developers no longer need to concern themselves with, for example, which application container to use, which version of Java, and which memory settings or garbage-collection (GC) policy to employ, they can simply push their applications to Cloud Foundry, and the applications run. This allows developers to focus on delivering applications that offer business value. Applications can then be bound to a wide set of backing services that are available on demand.

Units of Deployment

The phrase "the application is the unit of deployment" is used liberally. Applications as the sole unit of currency has changed with the emergence of Diego, Cloud Foundry's new runtime. Cloud Foundry now supports both applications running as long running processes (LRPs) and discrete "run once" tasks such as Bash scripts and Cron-like jobs. Diego LRPs are also referred to as application instances, or AIs. What you deploy, be it an actual app or just a script, is not important. The key takeaway is the removal of the need for deploying additional layers of technology.

Using cf push Command to Deploy

Cloud Foundry provides several ways for a user to interact with it, but the principal avenue is through its CLI. The most renowned CLI command often referenced by the Cloud Foundry community is `$ cf push`.

You use the `cf push` command to deploy your application. It has demonstrably improved the deployment experience. From the time you run `cf push` to the point when the application is available, Cloud Foundry performs the following tasks:

- Uploads and stores application files
- Examines and stores application metadata
- Stages the application by using a buildpack to create a droplet
- Selects an appropriate execution environment in which to run the droplet
- Starts the AI and streams logs to the Loggregator

This workflow is explored in more depth in Chapter 4.

Staging

Although it is part of the `cf push` workflow, staging is a core Cloud Foundry concept. Cloud Foundry allows users to deploy a prebuilt Docker image or an application artifact (source code or binaries) that has not yet been containerized. When deploying an application artifact, Cloud Foundry will *stage* the application on a machine or

VM known as a *Cell*, using everything required to compile and run the apps locally, including the following:

- The OS stack on which the application runs
- A buildpack containing all languages, libraries, dependencies, and runtime services the app uses

The staging process results in a droplet that the Cell can unpack, compile, and run. You can then run the resulting droplet (as in the case of a Docker image) repeatedly over several Cells. The same droplet runs the same app instances over multiple Cells without incurring the cost of staging every time a new instance is run. This ability provides deployment speed and confidence that all running instances from the same droplet are identical.

Self-Service Application Life Cycle

In most traditional scenarios, the application developer and application operator typically perform the following:

- Develop an application
- Deploy application services
- Deploy an application and connect (bind) it to application services
- Scale an application, both up and down
- Monitor an application
- Upgrade an application

This application life cycle is in play until the application is decommissioned and taken offline. Cloud Foundry simplifies the application life cycle by offering self-service capabilities to the end user. Adopting a self-service approach removes handoffs and potentially lengthy delays between teams. For example, the ability to deploy an application, provision and bind applications to services, scale, monitor, and upgrade are all offered by a simple call to the platform.

With Cloud Foundry, as mentioned earlier, the application or task itself becomes the single unit of deployment. Developers just push their applications to Cloud Foundry, and those applications run. If developers require multiple instances of an application to be running they can use `cf scale` to scale the application to *N* number of AIs. Cloud Foundry removes the cost and complexity of configuring infrastructure and middleware per application. Using a self-service model, users can do the following:

- Deploy applications

- Provision and bind additional services, such as messaging engines, caching solutions, and databases
- Scale applications
- Monitor application health and performance
- Update applications
- Delete applications

Deploying and scaling applications are completely independent operations. This provides the flexibility to scale at will, without the cost of having to redeploy the application every time. Users can simply scale an application with a self-service call to the platform. Through commercial products such as Pivotal Cloud Foundry, you can set up autoscaling policies for dynamic scaling of applications when they meet certain configurable thresholds.

Removing the infrastructure, OS, and middleware configuration concerns from developers allows them to focus all their effort on the application instead of deploying and configuring supporting technologies. This keeps the development focus where it needs to be, on the business logic that generates revenue.

The Twelve-Factor Contract

An architectural style known as cloud-native applications has been established to describe the design of applications specifically written to run in a cloud environment. These applications avoid some of the antipatterns that were established in the client-server era, such as writing data to the local filesystem. Those antipatterns do not work as well in a cloud environment because, for example, local storage is ephemeral given that VMs can move between different hosts. The Twelve-Factor App explains the 12 principles underpinning cloud-native applications.

Platforms offer a set of contracts to the applications and services that run on them. These contracts ensure that applications are constrained to do the right thing. *Twelve Factor* can be thought of as the contract between an application and a cloud-native platform.

There are benefits to adhering to a contract that constrains things correctly. Twitter is a great example of a constrained platform. You can write only 140 characters, but that constraint becomes an extremely valuable feature of the platform. You can do a lot with 140 characters coupled with the rich features surrounding that contract. Similarly, platform contracts are born out of previously tried-and-tested constraints; they are enabling and make doing the right thing—good developer practices—easy for developers.

Release Engineering through BOSH

In addition to developer concerns, the platform provides responsive IT operations, with full visibility and control over the application life cycle, provisioning, deployment, upgrades, and security patches. Several other operational benefits exist, such as built-in resilience, security, centralized user management, and better insights through capabilities like aggregated metrics and logging.

Rather than integrating, operating, and maintaining numerous individual components, the platform operator deals only with the platform. Structured platforms handle all the aforementioned undifferentiated heavy lifting tasks.

The Cloud Foundry repository is structured for use with BOSH. BOSH is an open source tool chain for release-engineering, deployment, and life cycle management. Using a YAML (YAML Ain't Markup Language) deployment manifest, BOSH creates and deploys (virtual) machines[1] on top of the targeted computing infrastructure and then deploys and runs software (in our case Cloud Foundry and supporting services) on to those created machines. Many of the benefits to operators are provided through using BOSH to deploy and manage Cloud Foundry. BOSH is often overlooked as just another component of Cloud Foundry, but it is the bedrock of Cloud Foundry and a vital piece of the ecosystem. It performs monitoring, failure recovery, and software updates with zero-to-minimal downtime.

Rather than utilizing a bespoke integration of a variety of tools and techniques that provide solutions to individual parts of the release-engineering goal, BOSH is designed to be a single tool covering the entire set of requirements of release engineering. BOSH enables software deployments to be:

- Automated
- Reproducible
- Scalable
- Monitored with self-healing failure recovery
- Updatable with zero-to-minimal downtime

BOSH translates intent into action via repeatability by always ensuring every provisioned release is identical and repeatable. This removes the challenge of configuration drift and removes the sprawl of snowflake servers.

BOSH configures infrastructure through code. By design, BOSH tries to abstract away the differences between infrastructure platforms (IaaS or physical servers) into

1 The terms VM and machine are used interchangeably because BOSH can deploy to multiple infrastructure environments ranging from containers to VMs, right down to configuring physical servers.

a generalized, cross-platform description of your deployment. This provides the benefit of being infrastructure agnostic (as far as possible).

BOSH performs monitoring, failure recovery, software updates, and patching with zero-to-minimal downtime. Without such a release-engineering tool chain, all these concerns remain the responsibility of the operations team. A lack of automation exposes the developer to unnecessary risk.

Built-In Resilience and Fault Tolerance

A key feature of Cloud Foundry is its built-in resilience. Cloud Foundry provides built-in resilience and self-healing based on *control theory*. Control theory is a branch of engineering and mathematics that uses feedback loops to control and modify the behavior of a dynamic system. Resiliency is about ensuring that the actual system state (the number of running applications, for example) matches the desired state at all times, even in the event of failures. Resiliency is an essential but often costly component of business continuity.

Cloud Foundry automates the recovery of failed applications, components, and processes. This *self-healing* removes the recovery burden from the operator, ensuring speed of recovery. Cloud Foundry, underpinned by BOSH, achieves resiliency and self-healing through:

- Restarting failed system processes
- Recreating missing or unresponsive VMs
- Deployment of new AIs if an application crashes or becomes unresponsive
- Application striping across availability zones (AZs) to enforce separation of the underlying infrastructure
- Dynamic routing and load balancing

Cloud Foundry deals with application orchestration and placement focused on even distribution across the infrastructure. The user should not need to worry about how the underlying infrastructure runs the application beyond having equal distribution across different resources (known as availability zones). The fact that multiple copies of the application are running with built-in resiliency is what matters.

Cloud Foundry provides dynamic load balancing. Application consumers use a route to access an application; each route is directly bound to one or more applications in Cloud Foundry. When running multiple instances, it balances the load across the instances, dynamically updating its routing table. Dead application routes are automatically pruned from the routing table, with new routes added when they become available.

Without these capabilities, the operations team is required to continually monitor and respond to pager alerts from failed apps and invalid routes. By replacing manual interaction with automated, self-healing software, applications and system components are restored quickly with less risk and downtime. The resiliency concern is satisfied once, for all applications running on the platform, as opposed to developing customized monitoring and restart scripts per application. The platform removes the ongoing cost and associated maintenance of bespoke resiliency solutions.

Self-Healing Processes

Traditional *infrastructure as code* tools do not check whether provisioned services are up and running. BOSH has strong opinions on how to create your release, forcing you to create a monitor script for the process. If a BOSH-deployed component has a process that dies, the monitor script will try to restart it.

Self-Healing VMs

BOSH has a Health Monitor and Resurrector. The Health Monitor uses status and life cycle events to monitor the health of VMs. If the Health Monitor detects a problem with a VM, it can trigger an alert and invoke the Resurrector. The Resurrector automatically recreates VMs identified by the Health Monitor as missing or unresponsive.

Self-Healing Application Instance Count

Cloud Foundry runs the application transparently, taking care of the application life cycle. If an AI dies for any reason (e.g., because of a bug in the application) or a VM dies, Cloud Foundry can self-heal by restarting new instances so as to keep the desired capacity to run AIs. It achieves this by monitoring how many instances of each application are running. The Cell manages its AIs, tracks started instances, and broadcasts state messages. When Cloud Foundry detects a discrepancy between the actual number of running instances versus the desired number of available AIs, it takes corrective action and initiates the deployment of new AIs. To ensure resiliency and fault tolerance, you should run multiple AIs for a single application. The AIs will be distributed across multiple Cells for resiliency.

Resiliency Through Availability Zones

Finally, Cloud Foundry supports the use of *availability zones* (AZs). As depicted in Figure 2-2, you can use AZs to enforce separation of the underlying infrastructure. For example, when running on AWS, you can directly map Cloud Foundry AZs to different AWS AZs. When running on vCenter, you can map Cloud Foundry AZs to different vCenter Cluster and resource-pool combinations. Cloud Foundry can then deploy its components across the AZs. When you deploy multiple AIs, Cloud Foun-

dry will distribute them evenly across the AZs. If, for example, a rack of servers fails and brings down an entire AZ, the AIs will still be up and serving traffic in the remaining AZs.

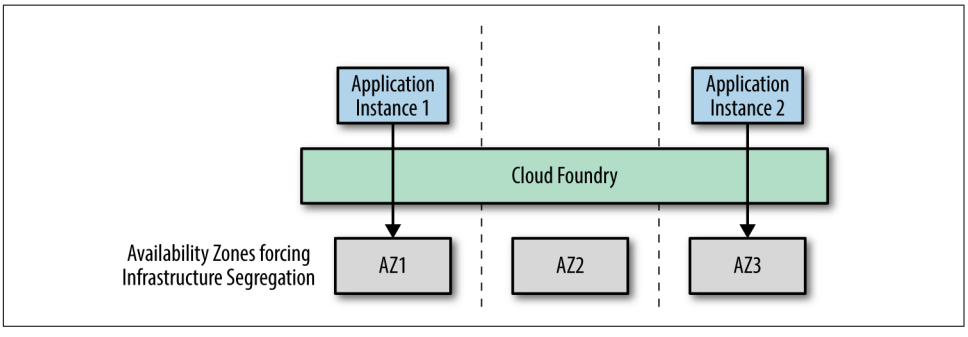

Figure 2-2. Application resiliency through Cloud Foundry AZs

Aggregated Streaming of Logs and Metrics

Cloud Foundry provides insight into both the application and the underlying platform through aggregated logging and metrics. The logging system within Cloud Foundry is known as the Loggregator. It is the inner voice of the system, telling the operator and developer what is happening. It is used to manage the performance, health, and scale of running applications and the platform itself, via the following:

- Logs provide visibility into behavior; for example, application logs can be used to trace through a specific call stack.
- Metrics provide visibility into health; for example, container metrics can include memory, CPU, and disk-per-app instance.

Insights are obtained through storing and analyzing a continuous stream of aggregated, time-ordered events from the output streams of all running processes and backing services. Application logs are aggregated and streamed to an endpoint via Cloud Foundry's Loggregator Firehose. Logs from the Cloud Foundry system components can also be made available and processed through a separate syslog drain. Cloud Foundry produces both the application and system logs to provide a holistic view to the end user.

Figure 2-3 illustrates how application logs and syslogs are separated as streams, in part to provide isolation and security between the two independent concerns, and in part due to consumer preferences. Generally speaking, app developers do not want to wade through component logs to resolve an app-specific issue. Developers can trace the log flow from the frontend router to the application code from a single log file.

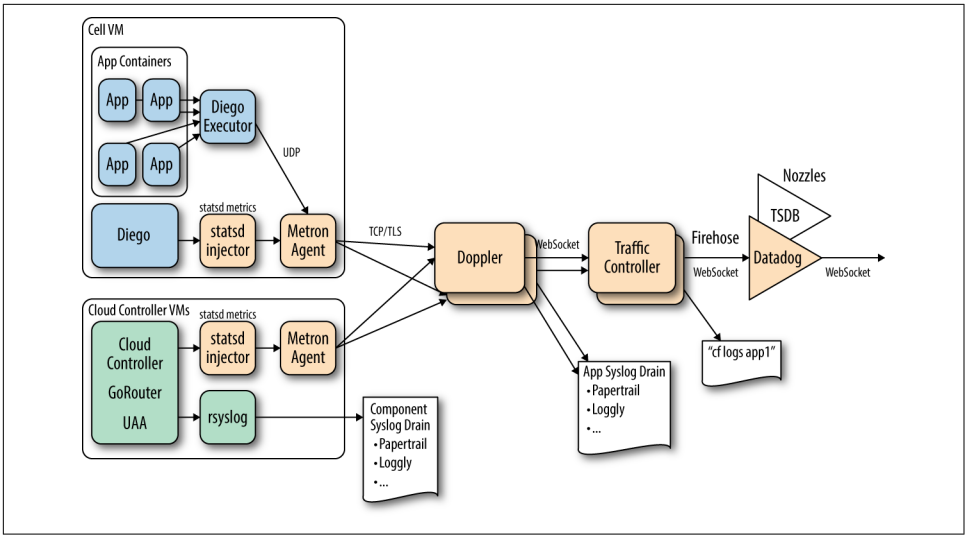

Figure 2-3. The Loggregator system architecture used for aggregating application logs and metrics

In addition to logs, metrics are gathered and streamed from system components. Operators can use metrics information to monitor an instance of Cloud Foundry. Furthermore, Cloud Foundry *events* show specific events such as when an application is started or stopped. The benefits of aggregated log, metric, and event streaming include the following:

- You can stream logs to a single endpoint.
- Streamed logs provide timestamped outputs per application.
- Both application logs and system-component logs are aggregated, simplifying their consumption.
- Metrics are gathered and streamed from system components.
- Operators can use metrics information to monitor an instance of Cloud Foundry.
- You can view logs from the command line or drain them into a log management service such as an ELK stack (Elasticsearch, Logstash, and Kibana), Splunk, or PCF Metrics.
- Viewing events is useful when debugging problems. For example, it is useful to be able to correlate an app instance event (like an app crash) to the container's specific metrics (high memory prior to crash).

The cost of implementing an aggregated log and metrics-streaming solution involves bespoke engineering to orchestrate and aggregate the streaming of both syslog and

application logs from every component within a distributed system into a central server. Using a platform removes the ongoing cost and associated maintenance of bespoke logging solutions.

Security

For enterprises working with cloud-based infrastructure, security is a top concern. Usually the security teams have the strongest initial objections to Cloud Foundry because it works in a way that is generally unprecedented to established enterprise security teams. However, as soon as these teams understand the strength of Cloud Foundry's security posture, my experience is that they become one of your strongest champions.

Distributed System Security

Cloud Foundry offers significant security benefits over traditional approaches to deploying applications because it allows you to strengthen your security posture once, for all applications deployed to the platform. However, securing distributed systems involves inherent complexity. For example, think about these issues:

- How much effort is required to automatically establish and apply network traffic rules to isolate components?
- What policies should be applied to automatically limit resources in order to defend against denial-of-service (DoS) attacks?
- How do you implement role-based access controls (RBAC) with in-built auditing of system access and actions?
- How do you know which components are potentially affected by a specific vulnerability and require patching?
- How do you safely patch the underlying OS without incurring application downtime?

These issues are standard requirements for most systems running in corporate data centers. The more custom engineering you use, the more you need to secure and patch that system. Distributed systems increase the security burden because there are more moving parts, and with the advances in container technology, new challenges arise, such as "How do you dynamically apply microsegmentation at the container layer?" Additionally, when it comes to rolling out security patches to update the system, many distributed systems suffer from configuration drift—namely, the lack of consistency between supposedly identical environments. Therefore, when working with complex distributed systems (specifically any cloud-based infrastructure), environmental risk factors are intensified.

The Challenge of Configuration Drift

Deployment environments (such as staging, quality assurance, and production) are often complex and time-consuming to construct and administer, producing the ongoing challenge of trying to manage configuration drift to maintain consistency between environments and VMs. Reproducible consistency through release-engineering tool chains such as Cloud Foundry's BOSH addresses this challenge.

Environmental Risk Factors for Advanced Persistent Threats

Malware known as advanced persistent threats (APTs) needs three risk factors in order to thrive:

1. Time
2. Leaked or misused credentials
3. Misconfigured and/or unpatched software

Given enough time, APTs can observe, analyze, and learn what is occurring within your system, storing away key pieces of information at will. If APTs obtain credentials, they can then further access other systems and data such as important ingress points into your protected data layer. Finally, unpatched software vulnerabilities provide APTs the freedom to further exploit, compromise, and expose your system.

Challenge of Minimal Change

There has been a belief that if enterprises deliver software with velocity, the trade-off is they must reduce their security posture and increase risk. Therefore, traditionally, many enterprises have relied on a concept of minimal change to mitigate risk and reduce velocity. Security teams establish strict and restrictive policies in an attempt to minimize the injection of new vulnerabilities. This is evident by ticketing systems to make basic configuration changes to middleware and databases, long-lived transport layer security (TLS) credentials, static firewall rules, and numerous security policies to which applications must adhere.

Minimal change becomes compounded by complexity of the environment. Because machines are difficult to patch and maintain, environmental complexity introduces a significant lag between the time a vulnerability is discovered and the time a machine is patched, be it months, or worse, even years in some production enterprise environments.

The Three Rs of Enterprise Security

IThese combined risk factors provide a perfect ecosystem in which APTs can flourish, and minimal change creates an environment in which all three factors are likely to occur. Cloud Foundry inverts the traditional enterprise security model by focusing on the three Rs of enterprise security: rotate, repave, repair.[2"]

1. Rotate the credentials frequently so that they are valid only for short periods of time.
2. Repave (rebuild) servers and applications from a known good state to cut down on the amount of time an attack can live.
3. Repair vulnerable software as soon as updates are available.

For the three Rs to be effective in minimizing the APT risk factors, you need to implement them repeatedly at high velocity. For example, data center credentials can be rotated hourly, servers and applications can be rebuilt several times a day, and complete vulnerability patching can be achieved within hours of patch availability. With this paradigm in mind, faster now equates to a safer and stronger security posture.

Additional Cloud Foundry Security

Cloud Foundry, along with BOSH and continuous integration (CI) tooling, provides tooling to make the three Rs' security posture a reality. Additionally, Cloud Foundry further protects you from security threats by automatically applying the following additional security controls to isolate applications and data:

- Manages software-release vulnerability by using new Cloud Foundry releases created with timely updates to address code issues
- Manages OS vulnerability by using a new OS created with the latest security patches
- Implements RBACs, applying and enforcing roles and permissions to ensure that users of the platform can view and affect only the resources to which they have been granted access
- Secures both code and the configuration of an application within a multitenant environment

2 The three Rs of enterprise security is a phrase coined in an article by Justin Smith, a cloud identity and security expert. I strongly suggest that if you're interested in enterprise security, you read the full article titled The Three R's of Enterprise Security.

- Deploys each application within its own self-contained and isolated containerized environment
- Prevents possible DoS attacks through resource starvation
- Provides an operator audit trail showing all operator actions applied to the platform
- Provides a user audit trail recording all relevant API invocations of an application
- Implements network traffic rules (security groups) to prevent system access to and from external networks, production services, and between internal components

BOSH and the underlying infrastructure expands the security posture further by handling data-at-rest encryption support through the infrastructure layer, usually by some device mechanism or filesystem-level support. For example, BOSH can use AWS EBS (Elastic Block Store) volume encryption for persistent disks.

Because every component within Cloud Foundry is created with the same OS image, Cloud Foundry eases the burden of rolling out these OS and software-release updates by using BOSH. BOSH redeploys updated VMs, component by component, to ensure zero-to-minimal downtime. This ultimately removes patching and updating concerns from the operator and provides a safer, more resilient way to update Cloud Foundry while keeping applications running. It is now totally possible to rebuild every VM in your data center from a known good state, as desired, with zero application downtime.

In addition, you can rebuild and redeploy the applications themselves from a known good release, upon request, with zero downtime. These rebuilding, repairing, and redeploying capabilities ensure that the patch turnaround time for the entire stack is as fast and as encompassing as possible, reaching every affected component across the stack with minimal human intervention. Cadence is limited only by the time it takes to run the pipeline and commit new code.

In addition to patching, if for any reason a component becomes compromised, you can instantly recreate it by using a known and clean software release and OS image, and move the compromised component into a quarantine area for further inspection.

There are additional detailed technical aspects that further improve security; for example, using namespaces for all containerized processes. I suggest reviewing the individual components for a more detailed understanding of how components such as Garden or the UAA help to further increase the security posture.

UAA Management

Role-based access defines who can use the platform and how. Cloud Foundry uses RBAC, with each role granting permissions to a specific environment the user is targeting. All collaborators target an environment with their individual user accounts associated with a role that governs what level and type of access the user has within that environment. Cloud Foundry's UAA is the central identity management service for both users and applications. It supports federated login, Lightweight Directory Access Protocol (LDAP), Security Assertion Markup Language (SAML), SSO, and multifactor authentication. UAA is a powerful component for strengthening your security posture for both user and application authentication.

Organizations and Spaces

Most developers are familiar with using VMs for development and deployment. Cloud Foundry is a virtualization layer (underpinned by containers) on top of a virtualization layer underpinned by VMs. Therefore, users do not have direct access to a specific machine or VM; rather, they simply access a logical partition of resources to deploy their apps.

To partition and allocate resources, Cloud Foundry uses logical boundaries known as Organizations (Orgs) and Spaces. Orgs contain one or more Spaces. Users can belong to any number of Orgs and/or Spaces, and users can have unique roles and permissions in each Org or Space to which they belong.

Orgs and Spaces provide the following:

- Logical separation and assignment of Cloud Foundry resources
- Isolation between different teams
- Logical isolation of development, test, staging, and production environments

For some enterprise customers with traditional silos, defining their required Orgs and Spaces can at first seem complex. Ideally, development teams should have autonomy to create and manage their own Spaces, as required. For development teams embracing microservices, the best approach is to organize teams by the big-A application—meaning a group of related (small-a) applications or services that can collectively and appropriately be grouped together, often referred to as *bulkheading*.

Ultimately, you know your business and how your developers work, so use these logical structures to provide meaningful working environments and pipelines for your developers.

Orgs

An *Org* is the top level of separation. A logical mapping could be to your business unit, a big-A application, or some other reasonable bounded context. When working with large enterprises that might have 200 developers in a business unit, I normally try to shift their thinking to the "Two-Pizza Team" model for application development. However, the actual number of developers within an Org or Space should not be a contentious point if it maps well to the physical organization and does not impede development or deployment.

Spaces

Every application and service is scoped to a Cloud Foundry *Space*. A Space provides a shared location that a set of users can access for application development, deployment, and maintenance. Every Space belongs to one Org. Each Org contains at least one Space but could contain several, and therefore can contain a broader set of collaborators than a single Space.

Environment Variables for Properties

The `cf push` command is the user saying to Cloud Foundry, "Here is my application artifact; run it on the cloud for me. I do not care how!"

The "I do not care how" needs explaining. There are properties you should care about, and these properties are configured by *environment variables*. Cloud Foundry uses environment variables to inform a deployed application about its environment. Environment variables include the following:

- How much memory to use
- What routes should be bound to the application
- How many instances of an application should be run
- Additional app-specific environment properties

The Cloud Foundry user can configure all of these properties.

Spaces are more of a developer concern. I believe there should be a limit to the amount of developers in a Space because there is a requirement for a level of trust due to the scope of shared resources and exposed environment variables that reside at the Space level. For the hyper security-conscious who have no trust between application teams, one Space per application is the only way forward. In reality, a Space for a big-A application might be more appropriate.

Colocation and Application Interactions

When considering the logical separations of your Cloud Foundry deployment, namely what Orgs and Spaces to construct, it is important to consider your application-to-application and application-to-services interactions. Although this consideration is more of a microservices consideration, an understanding of application and service boundaries is beneficial in understanding any colocation requirements. An example of this would be an application needing to access a corporate service in a specific data center or network. These concerns and their subsequent impacts become more noticeable at scale. You will need to design, understand, and document service discovery and app-to-app dependency, and additional frameworks such as Spring Cloud can significantly help here.

Resource Allocation

In addition to defining team structures, you can use Orgs and Spaces for assigning appropriate resources to each team.

Collaborators in an Org share the following:

- Resource quota
- Applications
- Services availability
- Custom domains

Domains Hosts and Routes

To enable traffic from external clients, applications require a specific URL known as a *route*. A route is a URL comprised of a domain and an optional host as a prefix. The host in this context is the portion of the URL referring to the application or applications, such as these:

my-app-name is the host prefix
my-business.com is the domain
my-app-name.my-business.com is the route

Route

Application consumers use a route to access an application. Each route is directly bound to one or more applications in Cloud Foundry. When running multiple instances, Cloud Foundry automatically load balances application traffic across mul-

tiple AIs through a component called the GoRouter. Because individual AIs can come and go for various reasons (scaling, app deletion, app crash), the GoRouter dynamically updates its routing table. Dead routes are automatically pruned from the routing table and new routes are added when they become available. Dynamic routing is a powerful feature. Traditional manual route table maintenance can be slow because it often requires submitting tickets to update or correct domain name server (DNS) or load balancer components.

Domains

Domains provide a namespace from which to create routes. Cloud Foundry uses domains within routes to direct requests to specific applications. You can also register and use a custom domain, known as an *owned domain*. Domains are associated with Orgs and are not directly bound to applications. Domains can be shared, meaning that they are registered to multiple Orgs, or private, registered to only one Org. Owned domains are always private.

Context Path–Based Routing

A context path in a URL extends a top-level route with additional context so as to route the client to either specific application functionality or a different application. For example, *http://my-app-name.my-business.com* can be extended to *http://my-app-name.my-business.com/home* to direct a client to a homepage.

In the preceding example, your clients can reach the application via *my-app-name.my-business.com*. Therefore, if a client targets that route and uses a different context path, it will still reach only a single application. For example, *http://my-app-name.my-business.com/home* and *http://my-app-name.my-business.com/somewhereelse* will both be routed by GoRouter to your app *my-app-name*.

This approach works if all the functionality under the route *my-app-name.my-business.com* can be served by a single app. However, when using microservices, there is a need to have a unified top-level route that can be backed by a number of microservices. Each service uses the same top-level domain but can be individually reached by different paths in the URL. The microservices collectively serve all supported paths under the domain *my-app-name.my-business.com*. With context path–based routing, you can independently scale those portions of your app that are experiencing high/low traffic.

Rolling Upgrades and Blue/Green Deployments

As discussed in "Security" on page 19, both the applications running on the platform and the platform itself allow for rolling upgrades and zero-downtime deployment through a distributed consensus.

You can update applications running on the platform with zero downtime through a technique known as blue/green deployments.

Summary

This chapter walked you through the principal concepts of Cloud Foundry. For example, you should now understand the meaning of a cloud OS, and the importance of the twelve-factor contract. The primary premise of Cloud Foundry is to enable the application development-to-deployment process to be as fast as possible. Cloud Foundry, underpinned by the BOSH release-engineering tool chain, achieves this by doing more on your behalf. Cloud Foundry provides the following:

- Built-in resiliency through automated recovery and self-healing of failed applications, components, and processes
- Built-in resiliency through striping applications across different resources
- Authentication and authorization for both users and applications, with the addition of RBAC for users
- Increased security posture with the ability to rotate credentials and repave and repair components
- The ability to update the platform with zero downtime via rolling upgrades across the system
- Speed in the deployment of apps with the ability to connect to a number of services via both platform-managed service brokers and services running in your existing IT infrastructure
- Built-in management and operation of services for your application, such as metrics and log aggregation, monitoring, autoscaling, and performance management

Now that you understand both the concepts and capabilities of Cloud Foundry, you are ready to learn about the individual components that comprise a Cloud Foundry deployment.

CHAPTER 3
Components

This chapter explores the details of Cloud Foundry components.

Cloud Foundry is a distributed system involving several different components. Distributed systems balance their processing loads over multiple networked machines.[1] They are optimized between efficiency and resiliency against failures. Cloud Foundry is comprised of a modular distributed architecture with discrete components utilized for dedicated purposes.

Distributed Systems

As a distributed system, each Cloud Foundry component has a well-defined responsibility. The different components interact with one another to achieve a common goal. Distributed systems achieve component interaction through communicating via messages and using central data stores for system-wide state coordination. There are several benefits to using a distributed component model, such as the ability to scale a single component in isolation, or the ability to change one component without directly affecting another.

It is important for the operator to understand what comprises the Cloud Foundry distributed system. For example, some components are responsible for system state and you need to back them up. Other components are responsible for running your applications, and therefore you most probably want more than one instance of those components to remain running to ensure resiliency. Ultimately, understanding these

1 The terms VM and machine are used interchangeably because BOSH can leverage and deploy to multiple infrastructure environments ranging from containers, to VMs, right down to configuring physical servers.

components and where their boundaries of responsibility lie is vital when it comes to establishing concerns such as resiliency and disaster recovery.

In this chapter, you will learn about the following:

1. The core components, including their purpose and boundary of responsibility
2. The flow of communication and interaction between components
3. The components responsible for state[2]

Component Overview

The Cloud Foundry components are covered in detail in the Cloud Foundry documentation. The Cloud Foundry code base is being rapidly developed by a growing open source team of around 250 developers. Any snapshot in time is going to change almost immediately. This book focuses not on the specific implementations that are subject to change, but on the purpose and function of each component. The implementation details change; the underlying patterns often remain.

We can group by function the components that make up Cloud Foundry into different layers. Table 3-1 lists these layers.

Table 3-1. Cloud Foundry component layers

Layer	Compoenents
Routing	GoRouter, TCPRouter, and external load balancer[a]
Authentication and user management	User Access and Authentication Management
Application life cycle and system state	Cloud Controller, Diego's core components (e.g., BBS and Brain)
App storage and execution	blobstore (including app artifacts/droplets and the Application Life-Cycle Binaries), Diego Cell (Garden, and runC)
Services	Service Broker, User Provided Service
Messaging	NATS (Network Address Translation) Messaging Bus
Metrics and logging	Loggregator (including Doppler and the Firehose)

[a] The external load balancer is not a Cloud Foundry component; it fronts the traffic coming into Cloud Foundry.

Figure 3-1 provides a visual representation of these components.

2 System state, along with configuration, is the most critical part of your environment; everything else is just wiring. Processes can come and go, but your system state and configuration must maintain integrity.

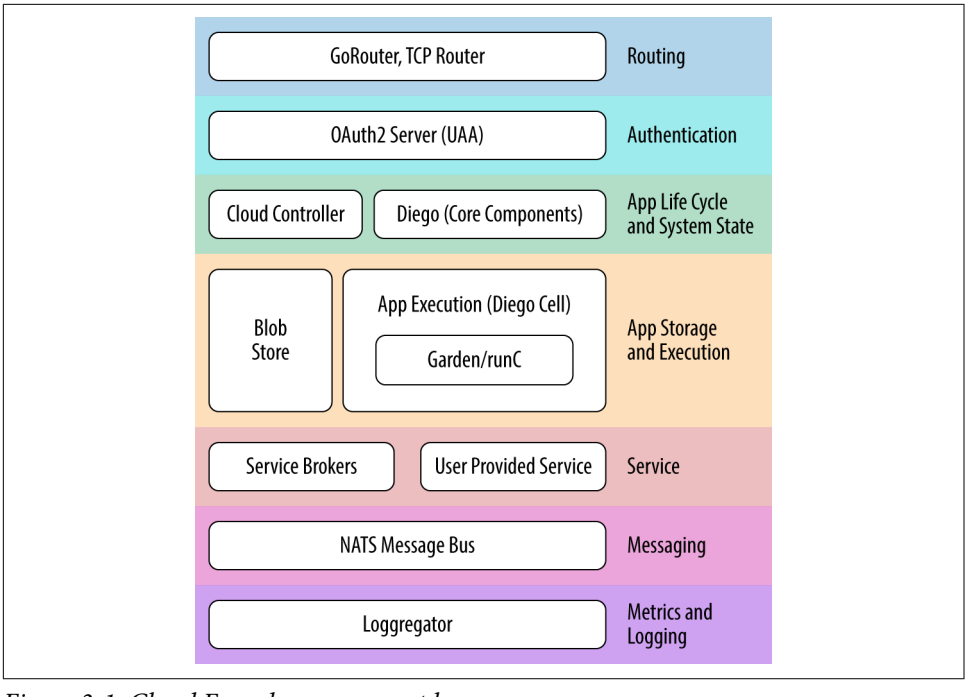

Figure 3-1. Cloud Foundry component layers

To discuss the core components and their role within Cloud Foundry, we will take a top-down approach beginning with components that handle traffic coming into Cloud Foundry.

Routing via the Load Balancer and GoRouter

All HTTP-based traffic first enters Cloud Foundry from an external load balancer fronting Cloud Foundry. The load balancer is primarily used for routing traffic to the GoRouter.

Load Balancer Preference

The choice of load balancer is according to your preference. Infrastructure hosted on AWS often use Amazon's elastic load balancer (ELB). On-premises deployments such as those on vSphere or OpenStack take advantage of existing enterprise load balancers such as F5's BIG-IP.

The load balancer can either handle secure socket layer (SSL) decryption and then route traffic on to the GoRouter, or pass the SSL connection on to the GoRouter for SSL decryption.

The GoRouter receives all incoming HTTP traffic from the load balancer. The term "router" can be misleading to networking engineers who expect routers to implement specific networking standards. Conceptually, the router should be treated as a reverse proxy, responsible for centrally managing and routing all incoming HTTP(S) traffic to the appropriate component. Traffic will typically be passed on to either an application or the Cloud Controller:

- Application users target their desired applications via a dedicated domain. The GoRouter will route application traffic to the appropriate AI running on a Diego Cell. If multiple AIs are running, the GoRouter will round-robin traffic across the AIs to distribute the workload.[3]
- Cloud Foundry users address the Cloud Controller: Cloud Foundry's API known as the CAPI. Some client traffic will go directly from the GoRouter to the UAA; however, most UAA calls are initiated from the Cloud Controller.

The GoRouter periodically queries Diego, Cloud Foundry's container runtime system, for information on the location of the Cells and containers on which each application is currently running.

Applications require a route for external traffic to access them. The GoRouter uses a routing table to keep track of the available applications. Because applications can have multiple AIs, all with a single route, each route has an associated array of host:port entries. The host is the Diego Cell machine running the application. The GoRouter regularly recomputes new routing tables based on the Cell's IP addresses and the host-side port numbers for the Cell's containers.

Routing is an important part of Cloud Foundry.

User Management and the UAA

As traffic enters Cloud Foundry, it needs to be authenticated. Cloud Foundry's UAA service is the central identity management service for managing:

- Cloud Foundry developers
- Application clients/end users
- Applications requiring application-to-application interactions

The UAA is an OAuth2 authorization server, that issues access tokens for client applications to use when they act on behalf of Cloud Foundry users; for example, when they request access to platform resources. The UAA is based on the most up-

3 The GoRouter supports Sticky Session configuration if required.

to-date security standards like OAuth, OpenID Connect, and System for Cross-domain Identity Management (SCIM). It authenticates platform users via their Cloud Foundry credentials. When users register an account with the Cloud Foundry platform, the UAA acts as the user identity store, retaining user passwords in the UAA database. The UAA can also act as a SSO service. It has endpoints for managing user accounts and registering OAuth2 clients as well as various other management functions. In addition, you can configure the UAA's user-identity store to either store user information internally or connect to an external user store through LDAP or SAML. Here are a couple of examples:

- Users can use LDAP credentials to gain access to the Cloud Foundry platform instead of registering a separate account.
- Operators can use SAML to connect to an external user store in order to enable SSO for users who want to access Cloud Foundry.

The UAA has its own database known as the UAADB. Like all databases responsible for system state, this is a critical component and you must make backups.

The Cloud Controller

The Cloud Controller exposes Cloud Foundry's REST API. Users of Cloud Foundry target the Cloud Controller so that they can interact with the Cloud Foundry API (CAPI). Clients directly interact with the Cloud Controller for tasks such as these:

- Pushing, staging, running, updating, and retrieving applications
- Pushing, staging, and running discrete one-off tasks

You can interact with the Cloud Controller in the following three ways:

- A scriptable CLI
- Language bindings (currently Java)
- Integration with development tools (IDEs) to ease the deployment process

You can find a detailed overview of the API commands at API Docs. The V3 API extends the API to also include Tasks in addition to applications LRPs.

The Cloud Controller is responsible for the System State and the Application Life-Cycle Policy.

System State

The Cloud Controller uses two components for storing state: a blobstore and a database known as the CCDB.

The Cloud Controller blobstore

To store large binary files such as application artifacts and staged application droplets, Cloud Foundry's Cloud Controller uses a blobstore that holds different types of artifacts:

- Application code packages—unstaged files that represent an application
- Resource files
- Buildpacks
- Droplets and other container images

Resource files are uploaded to the Cloud Controller and then cached in the blobstore with a unique secure hash algorithm (SHA) so as to be reused without reuploading the file. Before uploading all the application files, the Cloud Foundry CLI issues a *resource match* request to the Cloud Controller to determine if any of the application files already exist in the resource cache. When the application files are uploaded, the Cloud Foundry CLI omits files that exist in the resource cache by supplying the result of the resource-match request. The uploaded application files are combined with the files from the resource cache to create the complete application package.

Droplets are the result of taking an app package and staging it via a processing buildpack to create an executable binary artifact. The blobstore uses FOG so that it can use abstractions like Amazon Simple Storage Service (Amazon S3) or a mounted network file system (NFS) for storage.

The CCDB

The Cloud Controller, via its CCDB, also maintains records of the logical hierarchical structure including available orgs, spaces, apps, services, service instances, user roles, and more. It maintains users' information and how their roles map to orgs and spaces.

The Application Life-Cycle Policy

The Cloud Controller is responsible for the Application Life-Cycle Policy. Conceptually, the Cloud Controller is Cloud Foundry's CPU that drives the other components. For example, when you use the `cf push` command to push an application or task to Cloud Foundry, the Cloud Controller stores the raw application bits in its blobstore, creates a record to track the application metadata in its database, and directs the other system components to stage and run the application.

The Cloud Controller is application and task centric. It implements all of the object modeling around running applications (handling permissions, buildpack selection, service binding, etc.). The Cloud Controller is concerned with policy; for example,

"run two instances of my application," but the responsibility of orchestration and execution has been passed to Diego.

Continuous Delivery Pipelines

Many companies choose to interact with Cloud Foundry through a continuous delivery pipeline such as Concourse.ci that uses machines to utilize the CAPI. This approach reduces human error and offers deployment repeatability.

Application Execution

The components responsible for executing your applications and tasks include Diego, Garden (a container management API), and an Open Container Initiative (OCI)–compatible[4] backend container implementation (like runC).

Diego

Diego is the container runtime architecture for Cloud Foundry. Whereas the Cloud Controller is concerned with policy, it is Diego that provides the scheduling, orchestration, and placement of applications and tasks. Diego is designed to keep applications available by constantly monitoring their states and reconciling the actual system state with the expected state by starting and stopping processes as required. Chapter 4 covers Diego at length.

Garden and runC

Garden is a platform-agnostic Go API for container creation and management. Garden has pluggable backends for different platforms and runtimes, including Linux, Windows, and runC, an implementation of the OCI specification.

Metrics and Logging

As discussed in "Aggregated Streaming of Logs and Metrics" on page 17, system logs and metrics are continually generated and streamed from every component. In addition, AIs should produce logs as a continual stream for each running AI. Cloud Foundry's logging system is known as the Loggregator. The Loggregator aggregates component metrics and application logs into a central location using a Metron agent.

4 The OCI is an open governance structure for the express purpose of creating open industry standards around container formats and runtime. For more information, see *https://www.opencontainers.org/*.

Metron Agent

The Metron agent comes from the Loggregator subsystem and resides on every VM. It gathers a mix of metrics and log statistics from the Cloud Foundry components; for example, the Metron agent gathers application logs from the Cloud Foundry Diego hosts known as Cells. Operators can use the collected component logs and metrics to monitor a Cloud Foundry deployment. These metrics and logs can be forwarded by a syslog forwarder onto a syslog drain. It is possible to drain syslogs into multiple consumers. You can set up as many syslog "sinks" for an application as you want. Each sink will receive all messages.

Metron has the job of forwarding application logs and component metrics to the Loggregator subsystem by taking traffic from the various emitter sources (Cells in the case of apps) and routing that logging traffic to one or more Loggregator components. An instance of the Metron agent runs on each VM in a Cloud Foundry system and logs are therefore co-located on the emitter sources.

Loggregator

The Loggregator (log aggregator) system continually streams logging and metric information. The Loggregator's Firehose provides access to application logs, container metrics (memory, CPU, and disk-per-app instance), some component metrics, and component counter/HTTP events. If you want to see firsthand the output from the Loggregator, you can invoke the CF CLI command `$ cf logs APP`, as demonstrated in the following example:

```
2017-02-14T13:10:32.260-08:00 [RTR/0] [OUT] twicf-signup.cfapps.io -
[2017-02-14T21:10:32.250+0000] "GET /favicon.ico HTTP/1.1" 200 0 946
"https://twicf-signup.cfapps.io/"... "10.10.66.187:26208" "10.10.147
.77:60076" x_forwarded_for:"71.202.60.71" x_forwarded_proto:"https"
vcap_request_id:"2c28e2fe-54f7-48d7-5f0b-aaead5ab5c7c" response_time
:0.009104115 app_id:"ff073944-4d18-4c73-9441-f1a4c4bb4ca3" app_index:"0"
```

The Firehose does not provide component logs. Component logs are retrieved through an rsyslog drain.

Messaging

Most component machines communicate with one another internally through HTTP and HTTPS protocols. Temporary messages and data are captured in two locations:

- A Consul server stores the longer-lived control data such as component IP addresses and distributed locks that prevent components from duplicating actions.

- Diego's bulletin board system (BBS) stores a real-time view of the system, including the more frequently updated and disposable data such as Cell and application status, unallocated work, and heartbeat messages.

The Route-Emitter component still uses the NATS messaging system, a lightweight distributed publish–subscribe message queuing system, to broadcast the latest routing tables to the routers.

Additional Components

In addition to the aforementioned core system components, there are other components that comprise the Cloud Foundry ecosystem.

Stacks

A stack is a prebuilt root filesystem (rootfs). Stacks are used along with droplets (the output of buildpack staging). They provide the container filesystem used for running applications.

Cells can support more than one stack if configured correctly; however, a Cell must ensure buildpack and stack compatibility. For example, a Windows "stack" cannot run on Linux VMs. Therefore, to stage or run a Linux app, a Cell running a Linux stack must be available (and have free memory).

A Marketplace of On-Demand Services

Applications often depend on additional backing services such as databases, caches, messaging engines, or third-party APIs. Each Cloud Foundry deployment has the concept of a marketplace. The Cloud Foundry marketplace is a platform extension point. It exposes a set of services that are available for developers to use in support of running applications. Developers do not build applications in isolation. Applications often require additional middleware services such as data persistence, search, caching, graphing, messaging, API management, and more.

The platform operator exposes additional services to the marketplace through service brokers, route services, and user-provided services. The marketplace provides Cloud Foundry users with the self-service, on-demand provisioning of additional service instances. The platform operator can expose different service plans to different Cloud Foundry Orgs. Developers are able to view and create service instances only for service plans that have been configured to be visible for their targeted Org and Space. You can make service plans public and visible to all, or private to limit service visibility.

A service can offer different service plans to provide varying levels of resources or features for the same service. An example service plan is a database service offering

small, medium, or large plans with differing levels of concurrent connections and storage sizes. The provisioned service provides a unique set of credentials that can be used to bind and connect an application to the service.

Services not only enhance applications through providing middleware; they are concerned with all possible components to enable development teams to move quickly, including, for example, GitHub, Pivotal Tracker, CI services, and route services. For example, you can expose to the marketplace any application running on the platform that offers a service for others to consume. One advantage of this approach is that the service broker plans can prepopulate datastores with a specific schema or set of data (such as a sample customer set required for unit testing). Another example could be a service broker plan to provide specific preconfigured templates for apps.

Service brokers

Developers can provision service instances and then bind those instances to an application via a service broker responsible for providing the service instance.

A service broker interacts with the Cloud Controller to provision a service instance. Service brokers advertise a catalog of service offerings and service plans (e.g., a single-node MySQL plan or a clustered multinode MySQL plan). A service broker implements the CAPI to provide the user with the following:

- List service offerings
- Provision (create) and deprovision (delete) service instances
- Enable applications to bind to, and unbind from, the service instances

In general, *provision* reserves service resources (e.g., creates a new VM) and *bind* delivers the required information for accessing the resource to an application. The reserved resource is known, in Cloud Foundry parlance, as a *service instance*.

The service instance is governed by the broker author. What a service instance represents can vary not just by service, but also by plan. For example, an instance could be a container, a VM, or a new table and user in an existing database. Plans could offer a single database on a multitenant server, a dedicated datastore cluster, or simply an account and specific configuration on a running Cloud Foundry application. The key concern is that the broker implements the required API to interact with the Cloud Controller.

User-provided services

In addition to Cloud Foundry-managed service instances, operators can expose existing services via *user-provided services*. This allows established services such as a customer database to be bound to Cloud Foundry applications.

Buildpacks and Docker Images

How do you run applications inside Cloud Foundry? Applications can simply be *pushed* through the following CF CLI command:

```
$ cf push
```

There are currently two defined artifact types that you can `cf push`:

- A standalone application
- A prebuilt Docker image (that could contain additional runtime and middleware dependencies)

You can push standalone applications in both the form of a prebuild artifact such as a war/jar file or, in some cases, via raw source code such as a link to a Git remote.

Because a standalone application is not already part of a container image, when it is pushed to Cloud Foundry the buildpack process runs a compile phase. This involves compiling any source code and packaging the application and runtime dependencies into an executable container image. The buildpack process is also responsible for constructing the application runtime environment, deploying the application, and, finally, starting the required processes.

Buildpacks are a core link in the chain of the Cloud Foundry deployment process if you are deploying only an application artifact (e.g., JAR, WAR, Ruby, or Go source). The buildpack automates the following:

- The detection of an application framework
- The application compilation (known in Cloud Foundry terminology as *staging*)
- Running the application

The officially supported buildpacks are listed at *http://docs.cloudfoundry.org/buildpacks/index.html*. This list includes Ruby, Java (including other JVM-based languages), Go, Python, PHP, Node.js, and the binary and staticfile buildpacks.

Numerous additional community buildpacks exist. You can extend buildpacks or create new ones for specific language, framework, and runtime support. For example, the reason you might extend the Java buildpack (JBP) is to add support for additional application servers or a specific monitoring agent.

Buildpacks take your application artifact and containerize it into what Cloud Foundry calls a droplet. However, if you already have an OCI-compatible container image such as a Docker image, you can use `cf push` to move that directly to Cloud Foundry in order to run it. Containerization is the delivery mechanism for applications. This is true whether you push an application that Cloud Foundry containerizes into a droplet, or you push a prebuilt Docker image.

Infrastructure and the Cloud Provider Interface

Cloud Foundry relies on existing cloud-based infrastructure. The underlying infrastructure will have implementation-specific details. For example, vSphere's vCenter deals with clusters and resource pools, whereas AWS deals with regions and AZs. There are, however, some fundamental capabilities that need to be available and set up prior to installing Cloud Foundry:

- Networks and subnets (typically a /22 private network)
- VMs with specified CPU and memory requirements
- Storage for VMs
- File server or blobstore
- DNS, certificates, and wildcard domains
- Load balancer to pass traffic into the GoRouter
- NAT[5] for traffic flowing back to the load balancer

Cloud Foundry abstracts away infrastructure-specific implementations through the use of a cloud provider interface (CPI).

The Cloud Foundry GitHub Repository

Cloud Foundry uses the git system on GitHub to version-control all source code, documentation, and other resources such as buildpacks. Currently, the integrated Cloud Foundry code base can be located at cf-deployment. To check out Cloud Foundry code from GitHub, use the master branch because it points to the most recent stable final release.

Summary

By design, Cloud Foundry is a distributed system involving several components, grouped into the following functional layers:

- Routing for handling application and platform user traffic
- Authentication and user management
- Application life cycle and system state through the Cloud Controller and Diego's BBS
- Container runtime and app execution through Diego

5 NAT is only required if you are using nonroutable addresses.

- Services via the service marketplace
- Messaging
- Metrics and logging

Decoupling Cloud Foundry into a set of services allows each individual function to grow and evolve as required. Each Cloud Foundry component has a well-defined responsibility, and the different components interact with one another and share state in order to achieve a common goal. This loose coupling is advantageous:

- You can scale individual components in isolation
- You can swap and replace components to extend the platform capability
- You can promote a pattern of reuse

Now that you are aware of the components that comprise Cloud Foundry, you are ready to begin creating a Cloud Foundry instance.

CHAPTER 4

Diego

Diego is the container runtime architecture for Cloud Foundry. It is responsible for managing the scheduling, orchestration, and running of containerized workloads. Essentially, it is the heart of Cloud Foundry, running your applications and one-off tasks in containers, hosted on Windows and Linux backends. Most Cloud Foundry users (e.g., developers) do not interact with Diego directly. Developers interact only with Cloud Foundry's API, known as CAPI. However, comprehending the Diego container runtime is essential for Platform Operators because, as an operator, you are required to interact with Diego for key considerations such as resilience requirements and application troubleshooting. Understanding Diego is essential for understanding how workloads are deployed, run, and managed. This understanding also provides you with an appreciation of the principles underpinning container orchestration.

This chapter explores Diego's concepts and components. It explains the purpose of each Diego service, including how the services interrelate as state changes throughout the system.

Implementation Changes

It is important to understand the fundamental concepts of the Diego system. The specific technical implementation is less consequential because it is subject to change over time. What Diego does is more important than how it does it.

Why Diego?

Residing at the core of Cloud Foundry, Diego handles the scheduling, running, and monitoring of tasks and long-running processes (applications) that reside inside

managed containers. Diego extends the traditional notion of running applications to scheduling and running two types of processes:

Task
: A process that is guaranteed to be run at most once, with a finite duration. A Task might be an app-based script or a Cloud Foundry "job"; for example, a *staging request* to build an application droplet.

Long-running process (LRP)
: An LRP runs continuously and may have multiple instances. Cloud Foundry dictates to Diego the desired number of instances for each LRP, encapsulated as *DesiredLRPs*. All of these desired instances are run and represented as actual LRPs known as *ActualLRPs*. Diego attempts to keep the correct number of ActualLRPs running in the face of any network partitions, crashes, or other failures. ActualLRPs only terminate due to intervention, either by crashing or being stopped or killed. A typical example of an ActualLRP is an instance of an application running on Cloud Foundry.

Cloud Foundry is no longer solely about the *application as a unit of work*. The addition of running one-off tasks in isolation opens up the platform to a much broader set of workloads; for example, running Bash scripts or Cron-like jobs to process a one-off piece of data. Applications can also spawn and run tasks for isolated computation. Tasks can also be used for local environmental adjustments to ensure that applications adhere to service-level agreements (SLAs).

The required scope for what Cloud Foundry can run is ever increasing as more workloads migrate to the platform. Tasks and LRPs, along with the new TCPRouter, have opened up the platform to accommodate a much broader set of workloads. In addition to traditional web-based applications, you can now consider Cloud Foundry for the following:

- Internet of Things (IoT) applications such as aggregating and processing device data
- Batch applications
- Applications with application tasks
- Computational numerical modeling
- Reactive streaming applications
- TCP-based applications

By design, Diego is agnostic to preexisting Cloud Foundry components such as the Cloud Controller. This separation of concerns is compelling. Being agnostic to both client interaction and runtime implementation has allowed for diverse workloads

with composable backends. For example, Diego has generalized the way container image formats are handled; Diego's container management API is Garden.

The Importance of Container Management

For a container image (such as a Docker image or other OCI–compatible image) to be instantiated and run as an isolated process, it requires a container-management layer that can create, run, and manage the container process. The Docker company provides its own Linux Container manager for running Docker images. Other container-management technologies also exist, such as rkt and runC. As a more generalized container management API, Cloud Foundry uses Garden to support a variety of container technologies.

Through Garden, Diego can now support any container image format that the Garden API supports. Diego still supports the original droplet plus a stack combination but can now accommodate other image formats; for example, OCI-compatible images such as Docker or Rocket. In addition, Diego has added support for running containers on any Garden-based container technology, including Linux and Windows-based container backends that implement the Garden API. Figure 4-1 illustrates Diego's ability to support multiple application artifacts and container image formats.

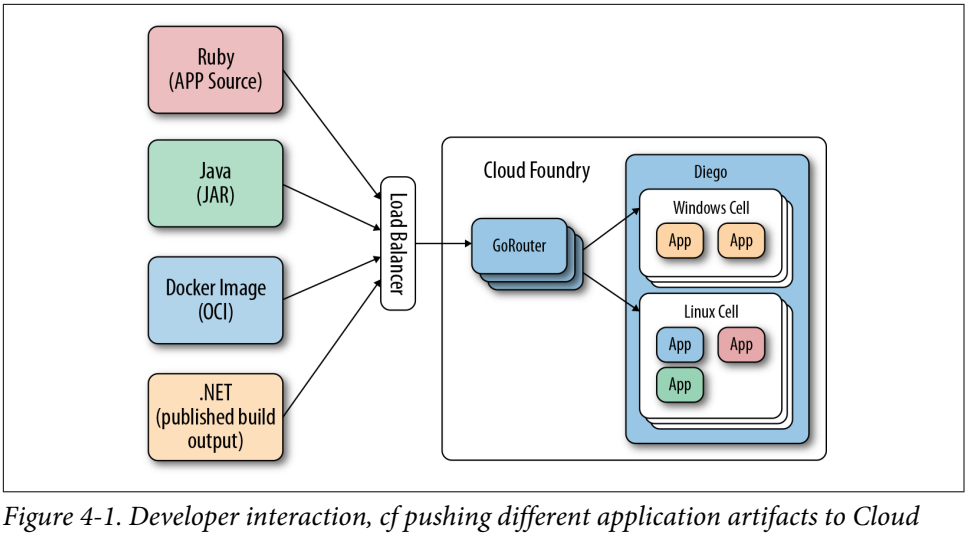

Figure 4-1. Developer interaction, cf pushing different application artifacts to Cloud Foundry

A Brief Overview of How Diego Works

Container scheduling and orchestration is a complex topic. Diego comprises several components, and each component comprises one or more microservices. Before getting into detail about these components and services, it is worth taking a moment to introduce the end-to-end flow of the Diego component interactions. At this point, I will begin to introduce specific component names for readability. The individual component responsibilities will be explained in detail in "Diego Components" on page 61.

At its core, Diego executes a scheduler. Scheduling is a method by which work, specified by some means, is assigned to resources that attempt to undertake and complete that work.

A Tip on Scheduling

Schedulers are responsible for using resources in such a way so as to allow multiple users to share system resources effectively while maintaining a targeted quality of service. Scheduling is an intrinsic part of the execution model of a distributed system. Scheduling makes it possible to have distributed multitasking spread over different nodes, with a centralized component responsible for processing. Within OSs, this processing unit is referred to as a CPU. Within Cloud Foundry, Diego acts as a centralized processing unit for all scheduling requirements.

Diego clients—in our case, the Cloud Controller (via a bridge component known as the CC-Bridge)—submit, update, and retrieve Tasks and LRPs to the BBS. The BBS service is responsible for Diego's central data store and API. Communication from the Cloud Controller to the BBS is via a remote procedure call (RPC)–style API implemented though Google protocol buffers.

The scheduler within Diego is governed by Diego's Brain component. The Brain's orchestration function, known as the Auctioneer service, retrieves information from the BBS and optimally distributes Tasks and LRPs to the cluster of Diego Cell machines (typically VMs). The Auctioneer distributes its work via an auction process that queries Cells for their capacity to handle the work and then sends work to the Cells. After the Auctioneer assigns work to a Cell, the Cell's Executor process creates a Garden container and executes the work encoded in the Task/LRP. This work is encoded as a generic, platform-independent recipe of composable actions (described in "Composable Actions" on page 49). Composable actions are the actual actions that run within a container; for example, a RunAction that runs a process in the container, or a DownloadAction that fetches and extracts an archive into the container. To assist in setting up the environment for the process running within the container,

Application Life-Cycle Binaries (e.g., Buildpacks) are downloaded from a file server that is responsible for providing static assets.

Application Life-Cycle Binaries

Staging is the process that takes an application and composes a executable binary known as a droplet. Staging is discussed further in "Staging" on page 87. The process of staging and running an application is complex and filled with OS and container implementation-specific requirements. These implementation-specific concerns have been encapsulated in a collection of binaries known collectively as the Application Life-Cycle. The Tasks and LRPs produced by the CC-Bridge download the Application Life-Cycle Binaries from a blobstore (in Cloud Foundry's case, the Cloud Controller blobstore). These Application Life-Cycle Binaries are helper binaries used to stage, start, and health-check Cloud Foundry applications. You can read more about Application Life-Cycle Binaries in "Application Life-Cycle Binaries" on page 76.

Diego ensures that the actual LRP (ActualLRP) state matches the desired LRP (DesiredLRP) state through interaction with its client (the CC-Bridge). Specific services on the CC-Bridge are responsible for keeping the Cloud Controller and Diego synchronized, ensuring domain freshness.[1]

The BBS also provides a real-time representation of the state of the Diego cluster (including all DesiredLRPs, running ActualLRP instances, and in-flight Tasks). The Brain's Converger periodically analyzes snapshots of this representation and corrects discrepancies, ensuring that Diego is eventually consistent. This is a clear example of Diego using a closed feedback loop to ensure that its view of the world is accurate. Self-healing is the first essential feature of resiliency; a closed feedback loop ensures that eventual consistency continues to match the ever-changing desired state.

Diego sends real-time streaming logs for Tasks/LRPs to the Loggregator system. Diego also registers its running LRP instances with the GoRouter via the Route-Emitter, ensuring external web traffic can be routed to the correct container.

Essential Diego Concepts

Before exploring Diego's architecture and component interaction, you need to be aware of two fundamental Diego concepts:

1 These services, along with the concept of domain freshness, are discussed further in "The CC-Bridge" on page 57.

- Action abstraction
- Composable actions

As work flows through the distributed system, Diego components describe their actions using different levels of abstraction. Diego can define different abstractions because the architecture is not bound by a single entity; for example, a single monolithic component with a static data schema. Rather, Diego consists of distributed components that each host one or more microservices. Although microservices architecture is no panacea, if designed correctly for appropriate use cases, microservices offer significant advantages by decoupling complex interactions. Diego's microservices have been composed with a defined boundary and are scoped to a specific component. Furthermore, Diego establishes well-defined communication flows between its component services. This is vital for a well designed system.

Components versus Services

Components like the Cell, Brain, and CC-Bridge are all names of VMs where Diego services are located. The location of those services is mutable, and depending on the deployment setup, some components can have more than one service residing on them. When describing the Diego workflow, consider the Diego components as though they were locations of specific services rather than the actual actors. For example, it makes sense to talk about distributing work to the Cells because a Cell is where the container running your workload will reside. However, the Cell does not actually create the container; it is currently the Rep service that is responsible for initiating container creation. Theoretically, the Rep could reside on some other component such as the BBS. Similarly, the Auctioneer is the service responsible for distributing work; the Brain is just the component VM on which the Auctioneer is running. In BOSH parlance, components are referred to as *instance groups* and services are referred to as *release jobs*.

Action Abstraction

Because each microservice is scoped to a specific Diego component, each service is free to express its work using its own abstraction. This design is incredibly powerful because bounded abstractions offer an unprecedented degree of flexibility. Abstraction levels move from course high-level abstractions to fine-grained implementations as work moves through the system. For example, work can begin its journey at the Cloud Controller as an app, but ultimately all work ends up as a scheduled process running within a created container. This low-level implementation of a *scheduled process* is too specific to be hardwired into every Diego component. If it were hardwired throughout, the distributed system would become incredibly complex for end

users and brittle to ongoing change. The abstraction boundaries provides two key benefits:

- The freedom of a plug-and-play model
- A higher-level concern for Diego clients

Plug-and-play offers the freedom to replace key parts of the system when required, without refactoring the core services. A great example of this is a pluggable container implementation. Processes can be run via a Docker image, a droplet plus a stack, or in a Windows container. The required container implementation (both image format and container backend) can be plugged into Diego as required without refactoring the other components.

Moreover, clients of Diego have high-level concerns. Clients should not need to be concerned with underlying implementation details such as how containers (or, more specifically, process isolation) are created in order to run their applications. Clients operate at a much higher level of abstraction, imperatively requesting *"run my application."* They are not required to care about any of the underlying implementation details. This is one of the core benefits of utilizing a platform to drive application velocity. The more of the undifferentiated heavy lifting you allow the platform to undertake, the faster your business code will progress.

Composable Actions

At the highest level of abstraction, the work performed by a Task or LRP is expressed in terms of composable actions, exposed via Diego's public API. As described earlier, composable actions are the actual actions that run within a container; for example, a RunAction that runs a process in the container, or a DownloadAction that fetches and extracts an archive into the container.

Conceptually each composable action implements a specific instruction. A set of composable actions can then be used to describe an explicit imperative action, such as *"stage my application"* or *"run my application."*[2] Composable actions are, by and large, hidden from Cloud Foundry users. Cloud Foundry users generally interact only with the Cloud Controller. The Cloud Controller (via the CC-Bridge) then interacts with Diego through Diego's composable actions. However, even though as a Platform Operator you do not interact with composable actions directly, it is essential that you understand the available composable actions when it comes to debugging Cloud Foundry. For example, the UploadAction might fail due to misconfigured blobstore credentials, or a TimeoutAction might fail due to a network partition.

2 The available actions are documented in the Cloud Foundry BBS Release on GitHub.

Composable actions include the following:

1. RunAction runs a process in the container.
2. DownloadAction fetches an archive (*.tgz* or *.zip*) and extracts it into the container.
3. UploadAction uploads a single file, in the container, to a URL via POST.
4. ParallelAction runs multiple actions in parallel.
5. CodependentAction runs multiple actions in parallel and will terminate all codependent actions after any single action exits.
6. SerialAction runs multiple actions in order.
7. EmitProgressAction wraps another action and logs messages when the wrapped action begins and ends.
8. TimeoutAction fails if the wrapped action does not exit within a time interval.
9. TryAction runs the wrapped action but ignores errors generated by the wrapped action.

Because composable actions are a high-level Diego abstraction, they describe generic activity, not how the activity is actually achieved. For example, the UploadAction describes uploading a single file to a URL; it does not specify that the URL should be the Cloud Controller's blobstore. Diego, as a generic execution environment, does not care about the URL; Cloud Foundry, as a client of Diego, is the entity responsible for defining that concern. This concept ties back to the action abstraction discussed previously, allowing Diego to remain as an independently deployable subsystem, free from specific end-user concerns.

So, how do composable actions relate to the aforementioned Cloud Foundry Tasks and LRPs? Consider the steps involved when the Cloud Controller issues a *run command* for an already staged application. To bridge the two abstractions, there is a Cloud Foundry-to-Diego bridge component known as the Cloud Controller Bridge (CC-Bridge). This essential function is discussed at length in "The CC-Bridge" on page 57. For now, it is sufficient to know that the CC-Bridge knows how to take the various resources (e.g., a droplet, metadata ,and blobstore location) that the Cloud Controller provides, coupled with a desired application message. Then, using these composable actions, the CC-Bridge directs Diego to build a sequence of composable actions to run the droplet within the container, injecting the necessary information provided by the Cloud Controller. For example, the specific composable action sequence for a *run command* will be:

- DownLoadAction to download the droplet from the CC blobstore into a specified location inside the container.

- DownLoadAction to download the set of static plugin (AppLifeCycle) binaries from a file server into a specified location within the container.
- RunAction to run the start command in the container, with the correct parameters. RunAction ensures that the container runs the code from the droplet using the helper (AppLifeCycle) binaries that correctly instantiate the container environment.

Applications are broken into a set of *Tasks* such as *"stage an app"* and *"run an app."* All Diego Tasks will finally result in a tree of composable actions to be run within a container.

Layered Architecture

Diego is comprised of a number of microservices residing on several components. Diego is best explained by initially referring to the diagram in Figure 4-2.

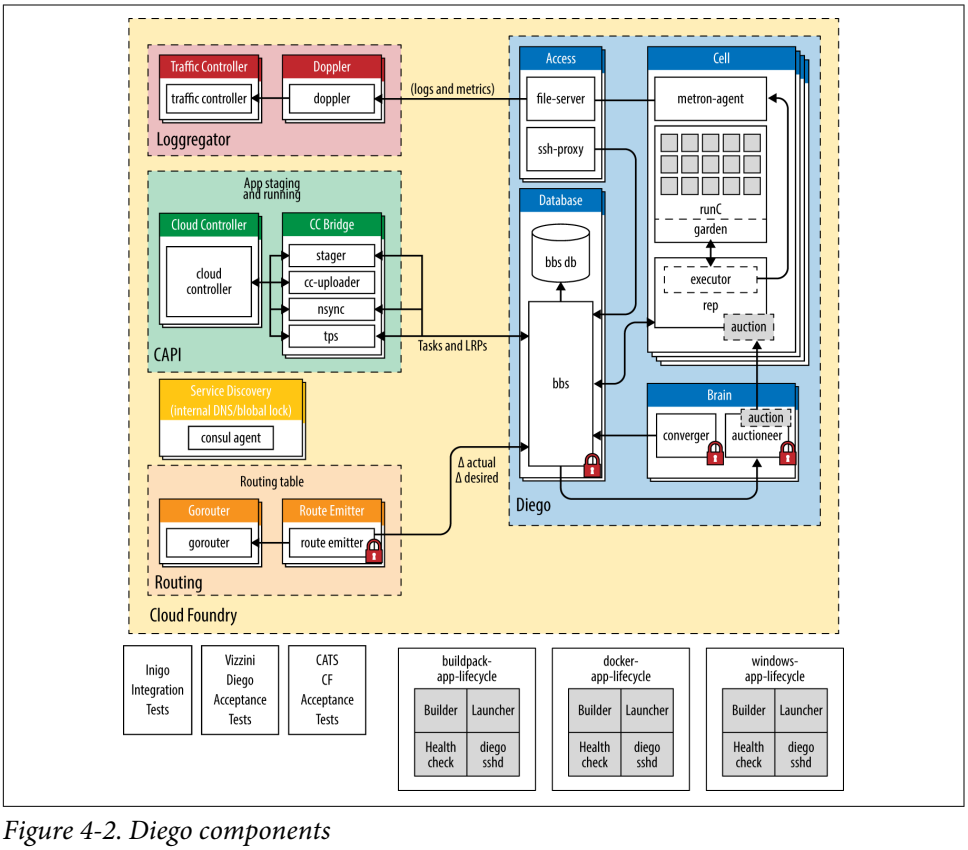

Figure 4-2. Diego components

We can group the components broadly, as follows:

- A Cloud Foundry layer of user-facing components (components with which you, as a platform user, will directly interact)
- The Diego Container Runtime layer (components with which the core Cloud Foundry components interact)

Each Diego component is a single deployable BOSH machine (known as an instance group) that can have any number of machine instances. Although there can be multiple instances of each instance group to allow for HA and horizontal scaling, some instance groups require a global lock to ensure that only one instance is allowed to make decisions.

Diego BOSH Releases

As with other Cloud Foundry components, you can deploy and run the Diego BOSH release as a standalone system. Having the property of being independently deployable is powerful. The engineering team responsible for the Diego subsystem has a smaller view of the distributed system allowing decoupling from other Cloud Foundry subsystems and faster iterative development. It also means that the release-engineering team can adopt more holistic granular testing in a heterogeneous environment of different subsystems. Because each Cloud Foundry subsystem can be deployed in isolation, more complex upgrade migration paths can now be verified because older versions of one subsystem, such as the Postgres release, can be deployed alongside the latest Diego release.[3]

We will begin exploring Diego by looking at the user-facing Cloud Foundry components that act as a client to Diego, and then move on to the specific Diego components and services.

Interacting with Diego

Cloud Foundry users do not interact with Diego directly; they interact with the Cloud Foundry user-facing components, which then interact with Diego on the user's behalf. Here are the Cloud Foundry user-facing components that work in conjunction with Diego:

- CAPI components: the Cloud Foundry API (Cloud Controller and the CC-Bridge)

3 You can find more information on Diego online, including the Diego BOSH release repository.

- The logging system defined by the Loggregator and Metron agents
- Routing (GoRouter, TCPRouter, and the Route-Emitter)

Collectively, these Cloud Foundry components are responsible for the following:

- Application policy
- Uploading application artifacts, droplets, and metadata to a blobstore
- Traffic routing and handling application traffic
- Logging
- User management including end-user interaction via the Cloud Foundry API commands[4]

Diego seamlessly hooks into these different Cloud Foundry components to run applications and tasks, route traffic to your applications, and allow the retrieval of required logs. With the exception of the CC-Bridge, these components were discussed at length in Chapter 3.

CAPI

Diego Tasks and LRPs are submitted to Diego via a Diego client. In Cloud Foundry's case, the Diego client is the CAPI, exposed by the Cloud Controller. Cloud Foundry users interact with the Cloud Controller through the CAPI. The Cloud Controller then interacts with Diego's BBS via the CC-Bridge, a Cloud Foundry-to-Diego translation layer. This interaction is depicted in Figure 4-3.

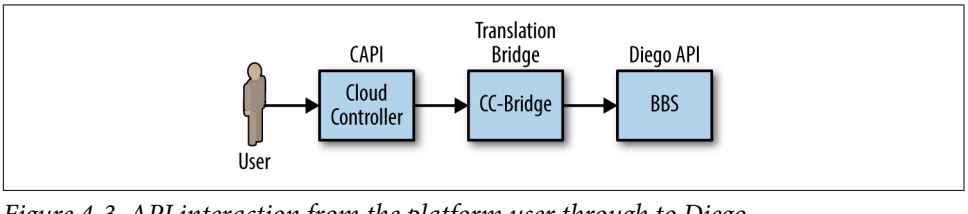

Figure 4-3. API interaction from the platform user through to Diego

The Cloud Controller provides REST API endpoints for Cloud Foundry users to interact with Cloud Foundry for commands including the following:

- Pushing, staging, running, and updating
- Pushing and running discrete one-off tasks

4 User management is primarily handled by the User Access and Authentication, which does not directly interact with Diego but still remains an essential piece of the Cloud Foundry ecosystem.

- Creating users, Orgs, Spaces, Routes, Domains and Services, and so on
- Retrieving application logs

The Cloud Controller (discussed in "The Cloud Controller" on page 33) is concerned with imperatively dictating policy to Diego, stating *"this is what the user desires; Diego, make it so!"*; for example, *"run two instances of this application."* Diego is responsible for orchestrating and executing the required workload. It deals with orchestration through a more autonomous subsystem at the backend. For example, Diego deals with the orchestration of the Cells used to fulfill a workload request through an auction process governed by the Auctioneer.

This design means that the Cloud Controller is not coupled to the execution machines (now known as Cells) that run your workload. The Cloud Controller does not talk to Diego directly; instead, it talks only to the translation component, the CC-Bridge, which translates the Cloud Controller's app-specific messages to the more generic Diego language of Tasks and LRPs. The CC-Bridge is discussed in-depth in "The CC-Bridge" on page 57. As just discussed, this abstraction allows the Cloud Foundry user to think in terms of apps and tasks, while allowing each Diego service to express its work in a meaningful abstraction that makes sense to that service.

Staging Workflow

To better understand the Cloud Controller interaction with Diego, we will explore what happens during a staging request. Exploring staging introduces you to two Diego components:

- The BBS: Diego's database that exposes the Diego API.
- Cells: the execution machines responsible for running applications in containers.

These two components are discussed at length later in this chapter. Understanding the staging process provides you with a clear picture of how Cloud Foundry interprets the `$ cf push` and translates it into a running ActualLRP instance. Figure 4-4 provides an overview of the process.

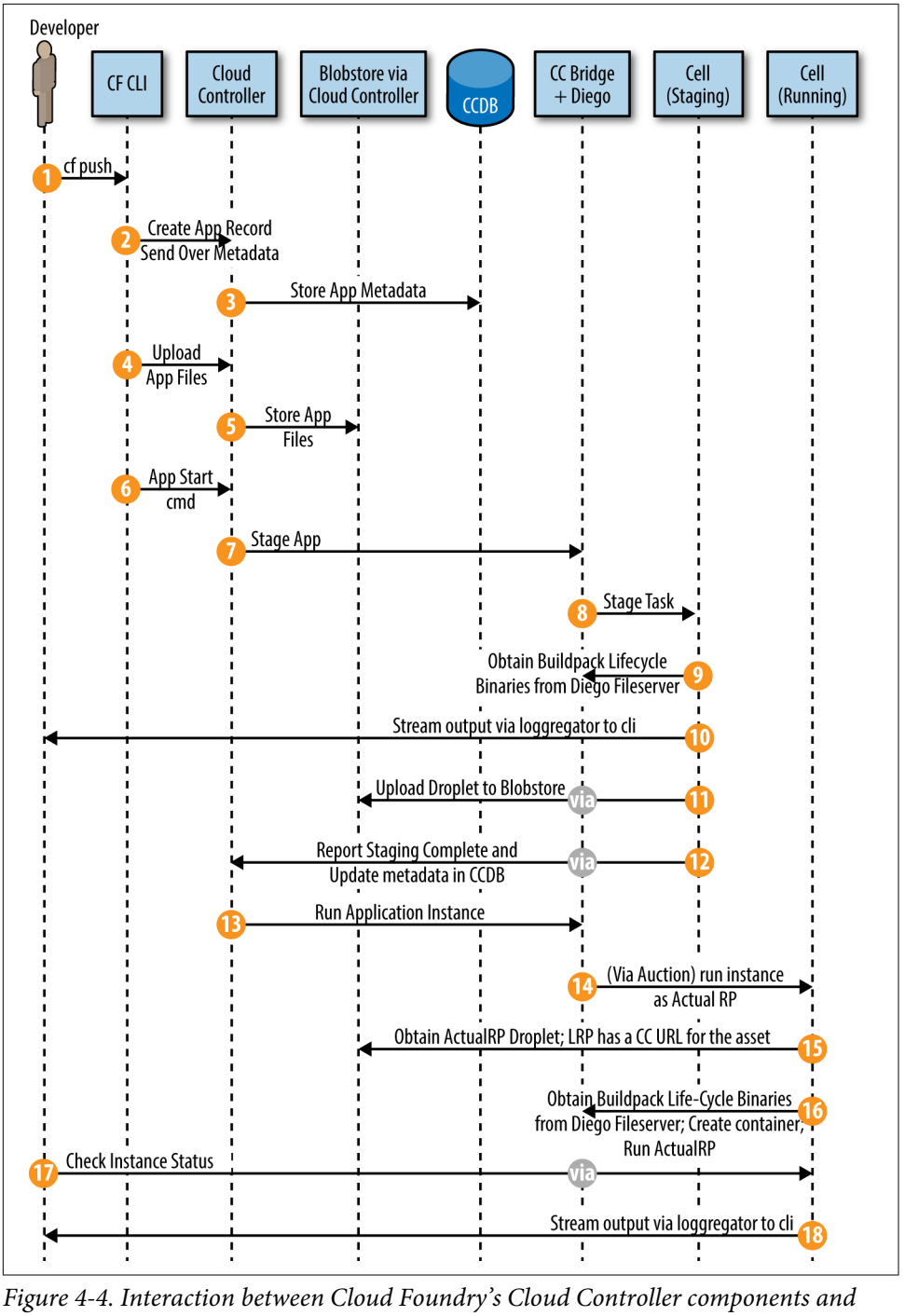

Figure 4-4. Interaction between Cloud Foundry's Cloud Controller components and Diego, while staging and running an application

The numbered list that follows corresponds to the callout numbers in Figure 4-4 and provides an explanation of each stage:

1. A developer/Platform Operator uses the Cloud Foundry command-line tool to issue a `cf push` command.
2. The Cloud Foundry command-line tool instructs the Cloud Controller to create a record for the application and sends over the application metadata (e.g., the app name, number of instances, and the required buildpack, if specified).
3. The Cloud Controller stores the application metadata in the CCDB.
4. The Cloud Foundry command-line tool uploads the application files (such as a *.jar* file) to the Cloud Controller.
5. The Cloud Controller stores the raw application files in the Cloud Controller blobstore.
6. The Cloud Foundry command-line tool issues an app start command (unless a `no-start` argument was specified).
7. Because the app has not already been staged, the Cloud Controller, through the CC-Bridge, instructs Diego to stage the application.
8. Diego, through an auction process, chooses a Cell for staging and sends the staging task to the Cell.
9. The staging Cell downloads the required life-cycle binaries that are hosted on the Diego file server, and then uses the instructions in the buildpack to run the staging task in order to stage the application.
10. The staging Cell streams the output of the staging process (to loggregator) so that the developer can troubleshoot application staging problems.
11. The staging Cell packages the resulting staged application into a tarball (*.tar* file) called a droplet and uploads it to the Cloud Controller blobstore.
12. Diego reports to the Cloud Controller that staging is complete. In addition, it returns metadata about the application back to the CCDB.
13. The Cloud Controller (via CC-Bridge) issues a run AI command to Diego to run the staged application.
14. Diego, through an auction process, chooses a Cell to run an LRP instance as an ActualLRP.
15. The running Cell downloads the application droplet directly from the Cloud Controller blobstore (the ActualLRP has a Cloud Controller URL for the asset).
16. The running Cell downloads the Application Life-Cycle Binaries hosted by the Diego file server and uses these binaries to create an appropriate container and then starts the ActualLRP instance.

17. Diego reports the status of the application to the Cloud Controller, which periodically receives the count of running instances and any crash events.
18. The Loggregator log/metric stream goes straight from the Cell to the Loggregator system (not via the CC-Bridge). The application logs can then be obtained from the Loggregator through the CF CLI.

Diego Staging

The preceding steps explore only staging and running an app from Cloud Foundry's perspective. The steps gloss over the interaction of the internal Diego services. After exploring the remaining Diego components and services, we will explore staging an LRP from Diego's perspective. Analyzing staging provides a concise way of detailing how each service interacts, and therefore it is important for you to understand.

The CC-Bridge

The CC-Bridge (Figure 4-5) is a special component comprised of four microservices. It is a translation layer designed to interact both with the Cloud Controller and Diego's API, which is exposed by Diego's BBS. The BBS is discussed shortly in "The BBS" on page 62.

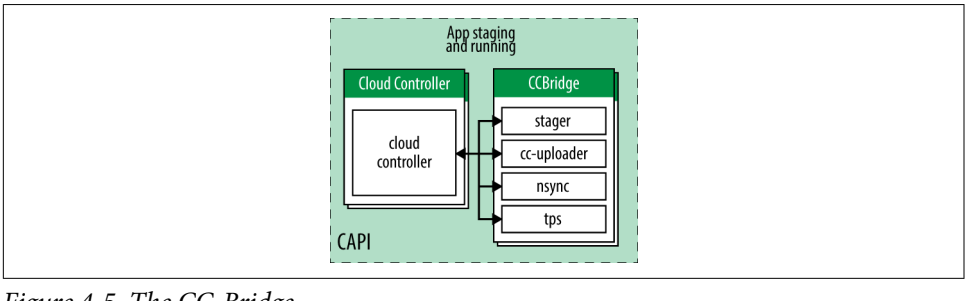

Figure 4-5. The CC-Bridge

The components on the CC-Bridge are essential for establishing *domain freshness*. Domain freshness means two things:

1. The actual state reflects the desired state.
2. The desired state is always understood.

Domain freshness is established through a combination of self-healing and closed feedback loops.

The Future of the CC-Bridge Component

For now, you can think of the CC-Bridge as a translation layer that converts the Cloud Controller's domain-specific requests into Diego's generic Tasks and LRPs. Eventually, Cloud Foundry's Cloud Controller might be modified to communicate directly with the BBS, making the CC-Bridge redundant. Either way, the function of the four services that currently reside on the CC-Bridge will still be required no matter where they reside.

CC-Bridge services translate the Cloud Controller's domain-specific requests of *stage* and *run* applications into Diego's generic language of LRP and Tasks. In other words, Diego does not explicitly know about applications or staging tasks; instead, it just knows about a Task or LRP that it has been requested to execute. The CC-Bridge services include the following:

Stager
: The Stager handles staging requests.

CC-Uploader
: A file server to serve static assets to the Cloud Controller's blobstore.

Nsync
: This service is responsible for handling domain freshness from the Cloud Controller to Diego.

TPS
: This service is responsible for handling domain freshness from Diego to the Cloud Controller.

Stager

The Stager handles staging requests from the Cloud Controller. It translates these requests into generic Tasks and submits the Tasks to Diego's BBS. The Stager also instructs the Cells (via BBS Task auctions) to inject the platform-specific Application Life-Cycle Binary into the Cell to perform the actual staging process. Application Life-Cycle Binaries provide the logic to build, run, and health-check the application. (You can read more about them in "Application Life-Cycle Binaries" on page 76.) After a task is completed (successfully or otherwise), the Stager sends a response to the Cloud Controller.

CC-Uploader

The CC-Uploader acts as a file server to serve static assets such as droplets and build artifacts to the Cloud Controller's blobstore. It mediates staging uploads from the Cell to the Cloud Controller, translating a simple generic HTTP POST request into

the complex correctly formed multipart-form upload request that is required by the Cloud Controller. Droplet uploads to the CC-Uploader are asynchronous, with the CC-Uploader polling the Cloud Controller until the asynchronous UploadAction is completed.

Nsync and TPS

Nsync and TPS are the two components responsible for handling domain freshness, matching desired and actual state between Diego and the Cloud Controller. They are effectively two sides of the same coin:

- The Nsync primarily retrieves information from the Cloud Controller. It is the component responsible for constructing the DesiredLRP that corresponds with the application request originating from the Cloud Controller.
- The TPS primarily provides feedback information from Diego to the Cloud Controller.

Both components react to events via their listener process, and will periodically check state validity via their respective bulker/watcher processes.

Figure 4-6 shows the high-level component interaction between the Cloud Controller, Diego's Nsync, and Converger components right through to the Cell. The figure also illustrates the convergence from a DesiredLRP stored in the BBS to an ActualLRP running on a Cell.

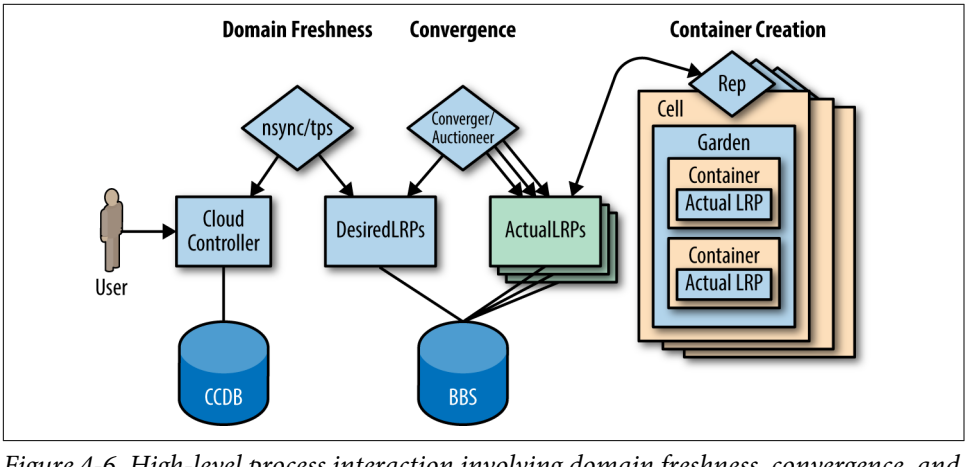

Figure 4-6. High-level process interaction involving domain freshness, convergence, and ActualLRP placement into a Cell's container

Domain Freshness

Diego's design has built-in resilience for its domain of responsibility—namely, running Tasks and LRPs. Diego's eventual-consistency model periodically compares the desired state (the set of DesiredLRPs) to the actual state (the set of ActualLRPs) and takes action to keep the actual and desired state synchronized. To achieve eventual consistency in a safe way, Diego must ensure that its view of the desired state is complete and up to date. Consider a rare scenario in which Diego's database has crashed and requires repopulating. In this context, Diego correctly knows about the ActualLRPs but has an invalid view of the DesiredLRPs, so it would be catastrophic for Diego to shut down all ActualLRPs in a bid to reconcile the actual and desired state.

To mitigate this effect, it is the Diego client's responsibility to repeatedly inform Diego of the desired state. Diego refers to this as the "freshness" of the desired state, or domain freshness. Diego consumers explicitly mark desired state as fresh on a domain-by-domain basis. Failing to do so will prevent Diego from taking actions to ensure eventual consistency (Diego will not stop extra instances if there is no corresponding desired state). To maintain freshness, the client typically supplies a *time-to-live* (TTL) and attempts to bump the freshness of the domain before the TTL expires (thus verifying that the contents of Diego's DesiredLRP are up to date). It is possible to opt out of this by updating the freshness with no TTL so that freshness will never expire, allowing Diego to always perform all of its eventual-consistency operations. Only destructive operations, performed during an eventual-consistency convergence cycle, will override freshness; Diego will continue to start and stop instances when explicitly instructed to do so.

The Nsync is responsible for keeping Diego *"in sync"* with the Cloud Controller. It splits its responsibilities between two independent processes: a bulker and a listener. Let's look at each of them:

Nsync-Listener
: The Nsync-Listener is a service that responds to DesiredLRP requests from the Cloud Controller. It actively listens for desired app requests and, upon receiving a request, either creates or updates the DesiredLRP via a record in the BBS database. This is the initial mechanism for dictating desired state from the Cloud Controller to Diego's BBS.

Nsync-Bulker
: The Nsync-Bulker is focused on maintaining the system's desired state, periodically polling the Cloud Controller and Diego's BBS for all DesiredLRPs to ensure that the DesiredLRP state known to Diego is up to date. This component pro-

vides a closed feedback loop, ensuring that any change of desired state from the Cloud Controller is reflected on to Diego.

The process status reporter (TPS) is responsible for reporting Diego's status; it is Diego's "hall monitor." It splits its responsibilities between two independent processes: listener and watcher submodules. Here's what each one does:

TPS-Listener
: The TPS-Listener provides the Cloud Controller with information about currently running ActualLRP instances. It responds to the Cloud Foundry CLI requests for `cf apps` and `cf app <may-app-name>`.

TPS-Watcher
: The TPS-Watcher monitors ActualLRP activity for crashes and reports them to the Cloud Controller.

Logging and Traffic Routing

To conclude our review of the Cloud Foundry layer of user-facing components, let look at logging.

Diego uses support for streaming logs from applications to Cloud Foundry's Loggregator system and provides support for routing traffic to applications via the routing subsystem. With the combined subsystems—Diego, Loggregator, and the routing subsystem—we have everything we need to do the following:

- Run any number of applications as a single user
- Route traffic to the LRPs
- Stream logs from the LRPs

The Loggregator aggregates and continually streams log and event data. Diego uses the Loggregator's Metron agent to provide real-time streaming of logs for all Tasks and LRPs in addition to the streaming of logs and metrics for all Diego components. The routing system routes incoming application traffic to ActualLRPs running within Garden containers on Diego Cells. (The Loggregator and routing subsystem were discussed in Chapter 3.)

Diego Components

At the time of writing, there are four core Diego component machines:

1. BBS (Database)
2. Cell
3. Brain

4. Access (an external component)

The functionality provided by these components is broken up into a number of microservices running on their respective component machines.

The BBS

The BBS manages Diego's database by maintaining an up-to-date cache of the state of the Diego cluster including a picture-in-time of all DesiredLRPs, running ActualLRP instances, and in-flight Tasks. Figure 4-7 provides an overview of the CC–Bridge-to-BBS interaction.

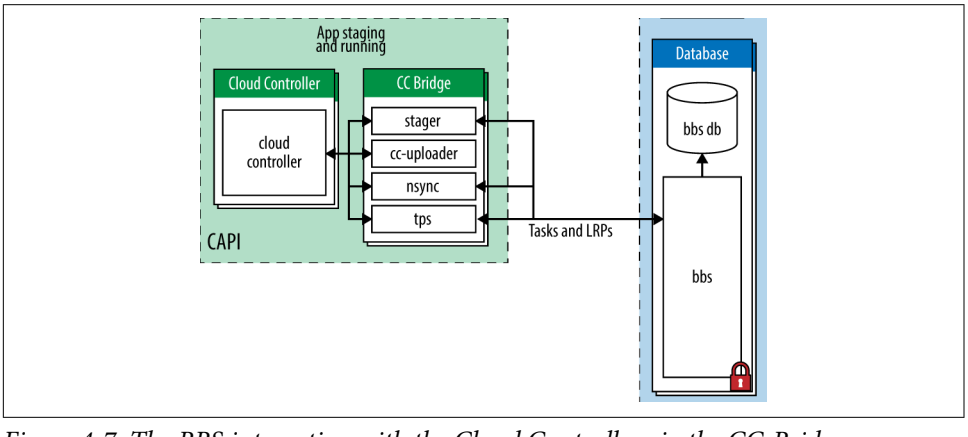

Figure 4-7. The BBS interaction with the Cloud Controller via the CC-Bridge

The BBS provides Diego's internal components and external clients with a consistent API to carry out the following:

1. Query and update the system's shared state (the state machine)
2. Trigger operations that execute the placement of Tasks and LRPs
3. View the datastore underpinning the state machine

For example, certain operations will cause effects. Consider creating a new LRP in the system. To achieve this, the CC-Bridge communicates to the BBS API endpoints. These endpoints will save any required state in the BBS database. If you specify a DesiredLRP with *N* instances, the BBS will automatically trigger the required actions for creating *N* ActualLRPs. This is the starting point of the state machine. Because understanding the state machine is important for debugging the platform, we cover it in more detail in "The Diego State Machine and Workload Life Cycles" on page 71.

Communication to the BBS is achieved through Google protocol buffers (protobufs).

Protocol Buffers

Protocol buffers (protobufs) provide an efficient way to marshal and unmarshal data. They are a language-neutral, platform-neutral, extensible mechanism for serializing structured data, similar in concept to eXtensible Markup Language (XML) but smaller, faster, and much less verbose.

The Diego API

The BBS provides an RPC-style API for core Diego components (Cell reps and Brain) and any external clients (SSH proxy, CC-Bridge, Route-Emitter). The BBS endpoint can be accessed only by Diego clients that reside on the same "private" network; it is not publicly routable. Diego uses dynamic service discovery between its internal components. Diego clients (the CC-Bridge) will look up the active IP address of the BBS using internal service discovery. Here are the main reasons why the Diego endpoint is not accessible via an external public route through the GoRouter:

- Clients are required to use TLS for communication with the BBS. The GoRouter is currently not capable of establishing or passing through a TLS connection to the backend.
- Clients are required to use mutual TLS authentication. To talk to the BBS, the client must present a certificate signed by a CA that both the client and BBS recognize.

The BBS encapsulates access to the backing database and manages data migrations, encoding, and encryption. The BBS imperatively directs Diego's Brain to state *"here is a task with a payload, find a Cell to run it."* Diego clients such as the CC-Bridge, the Brain, and the Cell all communicate with the BBS. The Brain and the Cell are both described in detail later in this chapter.

State versus Communication

The BBS is focused on data as opposed to communication. Putting state into an eventually consistent store (such as etcd, clustered MySQL, or Consul) is an effective way to build a distributed system because everything is globally visible. Eventual consistency is a data problem, not a communication problem. Tools like etcd and Consul, which operate as a quorum, handle consistency for you.

It is important that Diego does not store inappropriate data in a component designed for state. For instance, a "start auction" announcement is not a statement of truth; it is a transient piece of communication representing intent. Therefore, transient information should be communicated by messaging (i.e., NATS or HTTP—although direct communication over HTTPS is preferred). Communication via messaging is temporary and, as such, it can be lossy. For Diego auctions, loss is acceptable. Diego

can tolerate loss because of eventual consistency and the closed feedback loops described earlier.

For visibility into the BBS, you can use a tool called Veritas.

The Converger process

The Converger is a process responsible for keeping work eventually consistent.

Eventual Consistency

Eventual consistency is a consistency model used for maintaining the integrity of stateful data in a distributed system. To achieve high availability, a distributed system might have several "copies" of the data-backing store. Eventual consistency informally guarantees that, if no new updates are made to a given item of data, eventually all requests for that item will result in the most recently updated value being returned.

The Converger is a process that currently resides on the Brain instance group. It is important to discuss now, because it operates on the BBS periodically and takes actions to ensure that Diego attains eventual consistency. Should the Cell fail catastrophically, the Converger will automatically move the missing instances to other Cells. The Converger maintains a lock in the BBS to ensure that only one Converger performs convergence. This is primarily for performance considerations because convergence should be idempotent.

The Converger uses the converge methods in the runtime-schema/BBS to ensure eventual consistency and fault tolerance for Tasks and LRPs. When converging LRPs, the Converger identifies which actions need to take place to bring the DesiredLRP state and ActualLRP state into accord. Two actions are possible:

- If an instance is missing, a start auction is sent.
- If an extra instance is identified, a stop message is sent to the Cell hosting the additional instance.

In addition, the Converger watches for any potentially missed messages. For example, if a Task has been in the PENDING state for too long, it is possible that the request to hold an auction for the Task never made it to the Auctioneer. In this case, the Converger is responsible for resending the auction message. Periodically the Converger sends aggregate metrics about DesiredLRPs, ActualLRPs, and Tasks to the Loggregator.

Resilience with RAFT

Whatever the technology used to back the BBS (etcd, Consul, clustered MySQL), it is likely to be multinode to remove any single point of failure. If the backing technology is based on the Raft consensus algorithm, you should always ensure that you have an odd number of instances (three at a minimum) to maintain a quorum.

Diego Cell Components

Cells are where applications run. The term application is a high-level construct; Cells are concerned with running desired Tasks and LRPs. Cells are comprised of a number of subcomponents (Rep/Executor/Garden; see Figure 4-8) that deal with running and maintaining Tasks and LRPs. One Cell typically equates to a single VM, as governed by the CPI in use. You can scale-out Cells both for load and resilience concerns.

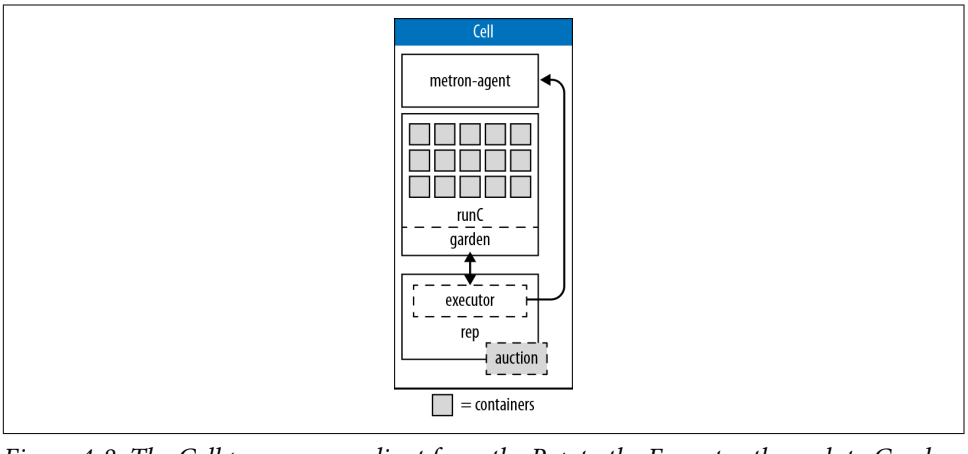

Figure 4-8. The Cell processes gradient from the Rep to the Executor through to Garden and its containers

There is a specificity gradient across the Rep, Executor, and Garden. The Rep is concerned with Tasks and LRPs, and knows the details about their life cycles. The Executor knows nothing about Tasks and LRPs but merely knows how to manage a collection of containers and run the composable actions in these containers. Garden, in turn, knows nothing about actions and simply provides a concrete implementation of a platform-specific containerization technology that can run arbitrary commands in containers.

Rep

The *Rep* is the Cell's API endpoint. It represents the Cell and mediates all communication between the BBS and Brain—the Rep is the only Cell component that commu-

nicates with the BBS. This single point of communication is important to understand; by using the Rep for all communication back to the Brain and BBS, the other Cell components can remain implementation independent. The Rep is also free to be reused across different types of Cells. The Rep is not concerned with specific container implementations; it knows only that the Brain wants something run. This means that specific container technology implementations can be updated or swapped out and replaced at will, without forcing additional changes within the wider distributed system. The power of this plug-and-play ability should not be underestimated. It is an essential capability for upgrading the system with zero-downtime deployments.

Specifically, the Rep does the following (see also Figure 4-9):

- Participates in the Brain's Auctioneer auctions to bid for Tasks and LRPs. It bids for work based on criteria such as its capacity to handle work, and then subsequently tries to accept what is assigned.
- Schedules Tasks and LRPs by asking its in-process Executor to create a container to run generic action recipes in the newly created container.
- Repeatedly maintains the presence of the Cell in the BBS. Should the Cell fail catastrophically, the BBS will invoke the Brain's Converger to automatically move the missing instances to another Cell with availability.
- Ensures that the set of Tasks and ActualLRPs stored in the BBS are synchronized with the active containers that are present and running on the Cell, thus completing the essential feedback loop between the Cell and the BBS.

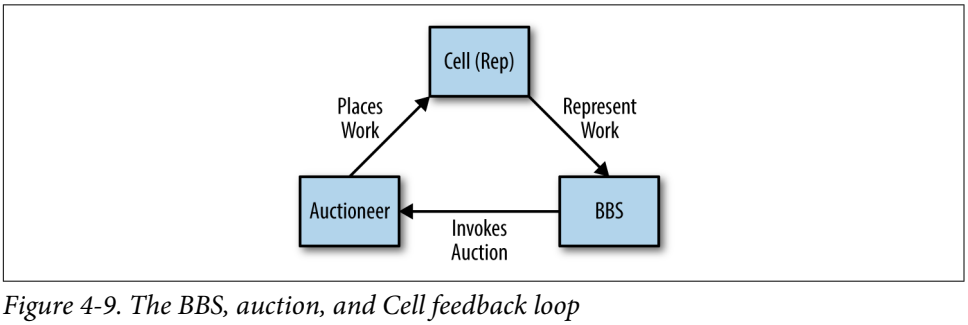

Figure 4-9. The BBS, auction, and Cell feedback loop

The Cell Rep is responsible for getting the status of a container periodically from Garden via its in-process Executor and reporting that status to the BBS database. There is only one Rep running on every Diego Cell.

Executor

The Executor is a logical process inside the Rep. Its remit is *"Let me run that for you."*[5] The Executor still resides in a separate repository, but it is not a separate job; it is part of the Rep.

The Executor does not know about the Task versus LRP distinction. It is primarily responsible for implementing the generic Executor "composable actions," as discussed in "Composable Actions" on page 49. Essentially, all of the translation between the Rep and Garden is encapsulated by the Executor: the Executor is a gateway adapter from the Rep to the Garden interface. The Rep deals with simplistic specifications (*execute this tree of composable actions*), and the Executor is in charge of actually interacting with Garden to, for example, make the ActualLRP via the life cycle objects. Additionally, the Executor streams Stdout and Stderr to the Metron-agent running on the Cell. These log streams are then forwarded to the Loggregator.

Garden

Cloud Foundry's container implementation, Garden, is separated into the API implementation and the actual container implementation. This separation between API and actual containers is similar to how Docker Engine has both an API and a container implementation based on runC.

Garden is a container API. It provides a platform-independent client server to manage Garden-compatible containers such as runC. It is backend agnostic, defining an interface to be implemented by container-runners (e.g., Garden-Linux, Garden-Windows, libcontainer, runC). The backend could be anything as long as it understands requests through the Garden API and is able to translate those requests into actions.

Container Users

By default, all applications run as the vcap user within the container. This user can be changed with a runAction, the composable action responsible for running a process in the container. This composable action allows you to specify, among other settings, the user. This means Diego's internal composable actions allow processes to run as any arbitrary user in the container. That said, the only users that really make sense are distinguished unprivileged users known as vcap and root. These two users are provided in the cflinuxfs2 rootfs. For Buildpack-based apps, Cloud Foundry always specifies the user to be vcap.

5 A conceptual adaption from the earlier container technology LMCTFY, which stands for *Let Me Contain That For You*.

The Diego Brain

We have already discussed the Brain's Converger process. The Brain (Figure 4-10) is also responsible for running the Auctioneer process. Auctioning is the key component of Diego's scheduling capability. There are two components in Diego that participate in auctions:

The Auctioneer
: Responsible for holding auctions whenever a Task or LRP needs to be scheduled

The Cell's Rep
: Represents a Cell in the auction by making bids for work and, if picked as the winner, running the Task or LRP

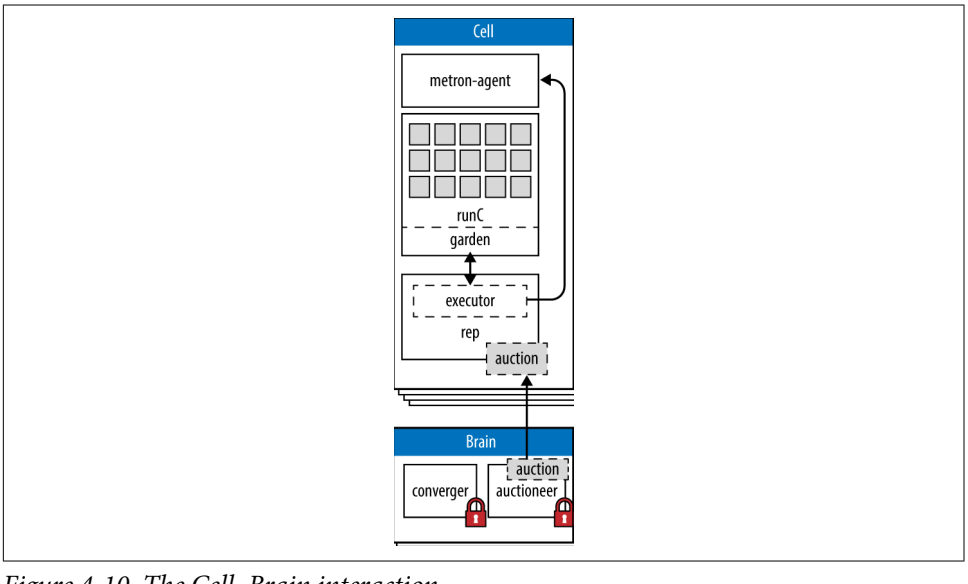

Figure 4-10. The Cell–Brain interaction

Horizontal Scaling for Controlling Instance Groups

Each core component (known in BOSH parlance as an instance group) is deployed on a dedicated machine or VM. There can be multiple instances of each instance group, allowing for HA and horizontal scaling.

Some instance groups, such as the Auctioneer, are essentially stateless. However, it is still important that only one instance is actively making decisions. Diego allows for lots of potential running instances in order to establish HA, but only one Auctioneer can be in charge at any one period in time. Within a specific instance group, the instance that is defined as "in charge" is specified by a global lock. Finding out which specific instance is in charge is accomplished through an internal service discovery mechanism.

When holding an auction, the Auctioneer communicates with the Cell Reps via HTTP. The auction process decides where Tasks and ActualLRP instances are run (remember that a client can dictate that one LRP requires several ActualLRP instances for availability). The Auctioneer maintains a lock in the BBS such that only one Auctioneer may handle auctions at any given time. The BBS determines which Auctioneer is active from a lock record (effectively the active Auctioneer holding the lock). When the BBS is at a point at which it wants to assign a payload to a Cell for execution, the BBS directs the Brain's Auctioneer by requesting, *"here is a task with a payload, find a Cell to run it."* The Auctioneer asks all of the Reps what they are currently running and what their current capacity is. Reps proactively bid for work and the Auctioneer uses the Reps' responses to make a placement decision.

Scheduling Algorithms

At the core of Diego is a distributed scheduling algorithm designed to orchestrate where work should reside. This distribution algorithm is based on several factors such as existing Cell content and app size. Other open source schedulers exist, such as Apache Mesos or Google's Kubernetes. Diego is optimized specifically for application and task workloads. The supply–demand relationship for Diego differs from the likes of Mesos. For Mesos, all worker Cells report, "*I am available to take* N *pieces of work*" and Mesos decides where the work goes. In Diego, an Auctioneer says, "*I have* N *pieces of work, who wants them?*" Diego's worker Cells then join an auction, and the winning Cell of each auctioned-off piece of work gets that piece of work. Mesos' approach is supply driven, or a "reverse-auction," and Diego's approach is demand driven.

A classic optimization problem in distributed systems is that there is a small lag between the time the system realizes it is required to make a decision (e.g., task place-

ment) and the time when it takes action on that decision. During this lag the input criteria that the original decision was based upon might have changed. The system needs to take account of this to optimize its work-placement decisions.

Consequently, there are currently two auction actions for LRPs: LRPStartAuction and LRPStopAuction. Let's look at each:

LRPStartAuctions
: These occur when LRPs need to be assigned somewhere to run. Essentially, one of Diego's Auctioneers is saying, *"We need another instance of this app. Who wants it?"*

LRPStopAuctions
: These occur when there are too many LRPs instances running for a particular application. In this case, the Auctioneer is saying, *"We have too many instances of this app at index X. Who wants to remove one?*"

The Cell that wins the auction either starts or stops the requested LRP.

Simulations

Diego includes simulations for testing the auction orchestration. Diego values the tight feedback loops achieved with the auction algorithms. This feedback allows the engineers to know how Diego internals are working and performing. It is especially valuable for dealing with order dependency: *where "A" must run before "B."* Simulating this means it can be tested and reasoned over more quickly.

Simulations can be run either in-process or across multiple processes. Unit tests are great for isolation and integration tests exercise the traditional usage. Simulation testing provides an extra layer of performance testing.

The Access VM

The access VM contains the file server and the SSH proxy services.

File server

The file server serves static assets used by various Diego components. In particular, it provides the Application Life-Cycle Binaries to the Cells.

The SSH proxy

Diego supports SSH access to ActualLRP instances. This feature provides direct access to the application for tasks such as viewing application logs or inspecting the state of the container filesystem. The SSH proxy is a stateless routing tier. The primary purpose of the SSH proxy is to broker connections between SSH clients and

SSH servers running within containers. The SSH proxy is a lightweight SSH daemon that supports the following:

- Command execution
- Secure file copy via SCP
- Secure file transfer via SFTP
- Local port forwarding
- Interactive shells, providing a simple and scalable way to access containers associated with ActualLRPs

The SSH proxy hosts the user-accessible SSH endpoint so that Cloud Foundry users can gain SSH access to containers running ActualLRPs. The SSH proxy is responsible for the following:

- SSH authentication
- Policy enforcement
- Access controls

After a user successfully authenticates with the proxy, the proxy attempts to locate the target container, creating the SSH session with a daemon running within the container. It effectively creates a "man-in-the-middle" connection with the client, bridging two SSH sessions:

- A session from the client to the SSH proxy
- A session from the SSH proxy to the container

After both sessions have been established, the proxy will manage the communication between the user's SSH client and the container's SSH daemon.

The daemon is self-contained and has no dependencies on the container root filesystem. It is focused on delivering basic access to ActualLRPs running in containers and is intended to run as an unprivileged process; interactive shells and commands will run as the daemon user. The daemon supports only one authorized key and is not intended to support multiple users. The daemon is available on Diego's file server. As part of the application life cycle bundle, Cloud Foundry's LRPs will include a downloadAction to acquire the binary and then a runAction to start it.

The Diego State Machine and Workload Life Cycles

Diego's semantics provide clients with the ability to state, *"Here is my workload: I want Diego to keep it running forever. I don't care how."* Diego ensures this request becomes a reality. If for some reason an ActualLRP crashes, Diego will reconcile

desired and actual state back to parity. These life cycle concerns are captured by Diego's state machine. It is important that you understand the state machine should you need to debug the system. For example, if you notice many of your ActualLRPs remain in UNCLAIMED state, it is highly likely that your Cells have reached capacity and require additional resources.

Stateful and Globally Aware Components

When it comes to state, Diego is comprised of three types of components: stateful components, stateless components, and stateless globally aware components. To describe the difference between the three components, consider the differences between the Cell, the BBS, and the Auctioneer.

A Cell has a global presence, but it is effectively stateless. Cells advertise themselves over HTTP to the BBS via a service discovery mechanism. They do not provide the entire picture of their current state; they simply maintain a presence in the BBS via recording a few static characteristics such as the AZ they reside in and their IP address. Maintaining a presence in the BBS allows the Auctioneer to contact the Cell at the recorded location in order to try to place work.

The BBS retains the global state associated with managing the persistence layer. The BBS also has the responsibility of managing migrations (between data migrations and schema migrations or API migrations reflected in the schema). For this reason, the BBS is an essential component to back up.

The Auctioneer is required to be globally aware; it has a global responsibility but is not directly responsible for system state. When a component has a global responsibility, there should only ever be one instance running at any one time.

An understanding of the state machine (see Figure 4-11) and how it relates to the app and task life cycles is essential for understanding where to begin with debugging a specific symptom.

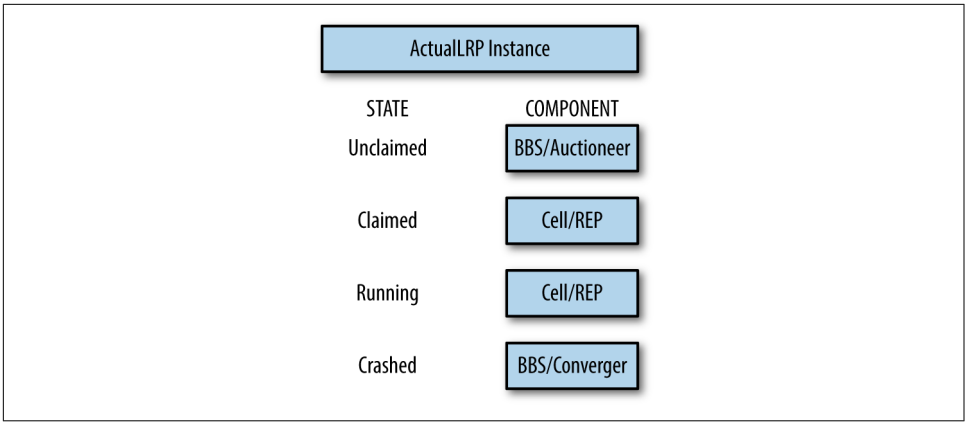

Figure 4-11. Diego's state machine

The responsibility for state belongs, collectively, to several components. The partition of ownership within the distributed system is dependent on the task or LRP (app) life cycle.

The Application Life Cycle

The application life cycle is as follows:

1. When a client expresses the desire to run an application, the request results in Diego's Nsync process creating an ActualLRP record.
2. An ActualLRP has a state field (a process globally unique identifier [GUID] with an index) recorded in the BBS. The ActualLRP begins its life in an UNCLAIMED state, resulting in the BBS passing it over to the Auctioneer process.
3. Work collectively is batched up and then distributed by the Auctioneer. When the Auctioneer has a batch of work that requires allocating, it looks up all the running Cells through service discovery and individually asks each Cell (Rep) for a current snapshot of its state, including how much capacity the Cell has to receive extra work. Auction activity is centrally controlled. It is like a team leader assigning tasks to team members based on their desire for the work and their capacity to perform it. The batch of work is broken up and distributed appropriately across the Cells.
4. An auction is performed and the ActualLRP is placed on a Cell. The Cell's Rep immediately transfers the ActualLRP state to CLAIMED. If placement is successful, the LRP is now RUNNING and the Rep now owns this record. If the ActualLRP cannot be placed (if a Cell is told to do work but cannot run that work for some reason), it reenters the auction process in an UNCLAIMED state. The Cell rejects the work and the Auctioneer can retry the work later.

5. The Cell's response on whether it has the capacity to do the work should not block actually starting the work. Therefore, the "perform" request from the Auctioneer instructs the Cell to try to undertake the work. The Cell then attempts to reserve capacity within the Executor's state management, identifying the extra resource required for the work.
6. The Cell quickly reports success or failure back to the Auctioneer. Upon success, the Cell then reserves this resource (as quickly as possible) so as not to advertise reserved resources during future auctions. The Executor is now aware that it has the responsibility for the reserved work and reports this back to the Rep, which knows the specifics of the Diego state machine for Tasks and LRPs.
7. Based on the current state that the Executor is reporting to the Rep, the Rep is then going to make decisions about how to progress workloads through their life cycles. In general, if anything is in reserved state, the Rep states to the Executor: "*start running that workload*." At that point, the Executor takes the container specification and creates a container in Garden.
8. The Auctioneer is responsible for the placement of workloads across the entire cluster of Cells. The Converger is responsible for making sure the desired and actual workloads are reconciled across the entire cluster of Cells. If the ActualLRP crashes, it is placed in a CRASHED state and the Rep moves the state ownership back to the BBS because the ActualLRP is no longer running. When the Rep undertakes its own local convergence cycle, trying to converge the actual running states in its Garden containers with its representation within the BBS, the Rep will discover the ActualLRP CRASHED state. The Rep then looks for the residual state that might still reside from its management of that ActualLRP. Even if the container itself is gone, the Executor might still have the container represented in a virtual state. The virtual state's "COMPLETED" state information might provide a clue as to why the ActualLRP died (e.g., it may have been placed in failure mode). The Rep then reports to the BBS that the ActualLRP has crashed. The BBS will attempt three consecutive restarts and then the restart policy will begin to back off exponentially, attempting subsequent restarts after a delayed period.
9. The Brain's Converger runs periodically looking for "CRASHED" ActualLRPs. Based on the number of times it has crashed (which is also retained in the BBS), the Converger will pass the LRP back to the Auctioneer to resume the ActualLRP. The Converger deals with the mass of unclaimed LRPs, moving them to ActualLRPs "CLAIMED" and "RUNNING". The Converger maps the state (held in the BBS) on to the desired LRPs. If the BBS desires five instances but the Rep only reports on four records, the Converger will make the fifth record in the BBS to kickstart the placement.
10. There is a spectrum of responsibilities that extends from the Converger to the BBS. Convergence requires a persistence-level convergence, including the

required cleanup process. There is also a business-model convergence in which we can strip away any persistence artifacts and deal with the concepts we are managing—and Diego ensures that the models of these concepts are in harmony. The persistence layer always happens in the BBS, but it is triggered by the Converger running a convergence loop.

Task Life Cycle

Tasks in Diego also undergo a life cycle. This life cycle is encoded in the Task's state as follows:

PENDING
: When a task is first created, it enters the PENDING state.

CLAIMED
: When successfully allocated to a Diego Cell, the Task enters the CLAIMED state and the Task's Cell_id is populated.

RUNNING
: The Task enters the RUNNING state when the Cell begins to create the container and run the defined Task action.

COMPLETED
: Upon Task completion, the Cell annotates the TaskResponse (failed, failure_reason, result), and the Task enters the COMPLETED state.

Upon Task completion, it is up to the consumer of Diego to acknowledge and resolve the completed Task, either via a completion callback or by deleting the Task. To discover if a Task is completed, the Diego consumer must either register a completion_callback_url or periodically poll the API to fetch the Task in question. When the Task is being resolved, it first enters the RESOLVING state and is ultimately removed from Diego. Diego will automatically reap Tasks that remain unresolved after two minutes.

Additional Components and Concepts

In addition to its core components, Diego also makes use of the following:

- The Route-Emitter
- Consul
- Application Life-Cycle Binaries

The Route-Emitter

The Route-Emitter is responsible for registering and unregistering the ActualLRPs routes with the GoRouter. It monitors DesiredLRP state and ActualLRP state via the information stored in the BBS. When a change is detected, the Route-Emitter emits route registration/unregistration messages to the router. It also periodically emits the entire routing table to the GoRouter.

Consul

Consul is a highly available and distributed service discovery and key-value store. Diego uses it currently for two reasons:

- It provides dynamic service registration and load balancing via internal DNS resolution.
- It provides a consistent key-value store for maintenance of distributed locks and component presence. For example, the active Auctioneer holds a distributed lock to ensure that other Auctioneers do not compete for work. The Cells Rep maintains a global presence in Consul so that Consul can maintain a correct view of the world. The Converger also maintains a global lock in Consul.

To provide DNS, Consul uses a cluster of services. For services that require DNS resolution, a Consul agent is co-located with the hosting Diego component. The consul-agent job adds 127.0.0.1 as the first entry in the nameserver list. The consul-agent that is co-located on the Diego component VM serves DNS for consul-registered services on 127.0.0.1:53. When Consul tries to resolve an entry, the Consul domain checks 127.0.0.1 first. This reduces the number of component hops involved in DNS resolution. Consul allows for effective intercomponent communication.

Other services that expect external DNS resolution also need a reference to the external DNS server to be present in */etc/resolv.conf*.

Like all RAFT stores, if Consul loses quorum, it may require manual intervention. Therefore, a three-Consul-node cluster is required, preferably spanning three AZs. If you restart a node, when it comes back up, it will begin talking to its peers and replay the RAFT log to get up to date and synchronized with all the database history. It is imperative to ensure that a node is fully back up and has rejoined the cluster prior to taking a second node down; otherwise, when the second node goes offline, you might lose quorum. BOSH deploys Consul via a rolling upgrade to ensure that each node is fully available prior to bringing up the next.

Application Life-Cycle Binaries

Diego aims to be platform agnostic. All platform-specific concerns are delegated to two types of components:

- the Garden backend
- the Application Life-Cycle Binaries

The process of staging and running an application is complex. These concerns are encapsulated in a set of binaries known collectively as the *Application Life-Cycle Binaries*. There are different Application Life-Cycle Binaries depending on the container image (see also Figure 4-12):

- Buildpack-Application Life Cycle implements a traditional buildpack-based life cycle.
- Docker-Application Life Cycle implements a Docker-based OCI-compatible life cycle.
- Windows-Application Life Cycle implements a life cycle for .NET applications on Windows.

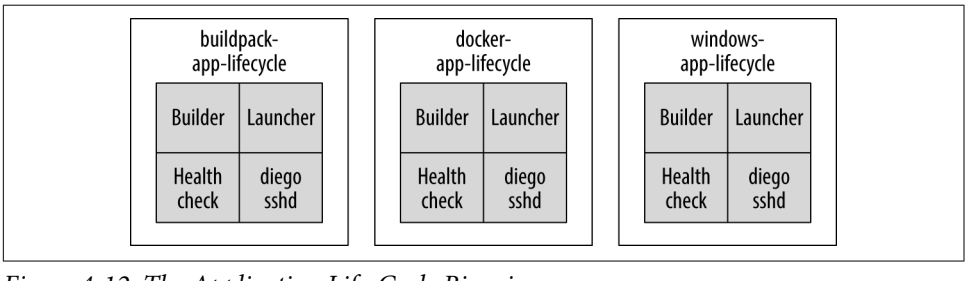

Figure 4-12. The Application Life-Cycle Binaries

Each of the aforementioned Application Life Cycles provides a set of binaries that manage a specific application type. For the Buildpack-Application Life Cycle, there are three binaries:

- The Builder stages a Cloud Foundry application. The CC-Bridge runs the Builder as a Task on every staging request. The Builder performs static analysis on the application code and performs any required preprocessing before the application is first run.
- The Launcher runs a Cloud Foundry application. The CC-Bridge sets the Launcher as the Action on the app's DesiredLRP. The Launcher executes the user's start command with the correct system context (working directory, environment variables, etc.).
- The Healthcheck performs a status check of the running ActualLRP from within the container. The CC-Bridge sets the Healthcheck as the Monitor action on the app's DesiredLRP.

The Stager Task produced by the CC-Bridge downloads the appropriate Application Life-Cycle Binaries and runs them to invoke life cycle scripts such as *stage*, *start*, and *health-check* in the ActualLRP.

This is a pluggable module for running OS-specific components. Because the life cycle is OS specific, the OS is an explicitly specified field required by the LRP. For example, the current Linux setting references the cflinuxfs2 rootfs. In addition to the rootfs, the only other OS-specific component that is explicitly specified is the back-end container type.

RunAction and the Launcher Binary

When Diego invokes `start-command` for an app (ActualLRP), it does so as a RunAction. A Task or LRP RunAction invokes a process that resides under the process identifier (PID) namespace of the container within which it is invoked. The RunAction process is not like a simple `bash -c "start-command"`. It is run through the launcher binary, which adds additional environmental setup for the process before the process calls `exec` to spawn `bash -c "start-command"`. The launcher is invoked after the container is created and is focused on setting up the environment for the `start-command` process (e.g., the launcher sets appropriate environmental variables, the working directory, and all other environment concerns inside the container). This environment setup is not very extensive; for example, for buildpack apps, it changes HOME and TMPDIR, puts some per-instance information in VCAP_APPLICATION, and then sources any scripts the buildpack placed in *.profile.d*. When the launcher runs, it uses the `exec` system call to invoke the start command (and so disappears). When you review the process tree within the Garden container, you can observe that after the launcher finishes running and setting up the environment, the top-level process is `init` (the first daemon process started when booting the system), and your applications will be a child of `initd`.

In addition to the Linux life cycle, Diego also supports a Windows life cycle and a Docker life cycle. The Docker life cycle (to understand how to stage a Docker image) is based on metadata from the Cloud Controller, based on what type of app we want to run. As part of the information for *"stage these app bits,"* there will be some indication of what branching the Stager is required to undertake. The binaries have tried to ensure that, as much as possible, these specifications are shared.

Putting It All Together

We discussed what happens when staging an application in "Staging Workflow" on page 54. Specifically, we discussed the following interactions:

- The CLI and the Cloud Controller

- The Cloud Controller and CC-Bridge
- The CC-Bridge and Diego, including Diego Cells

Up to this point, we have glossed over the interaction of the internal Diego components. This section discusses the interaction between the Diego components during staging LRPs. This section assumes you are staging an app using the buildpack Application Life-Cycle Binaries as opposed to pushing a prebuilt OCI-compatible image.

Staging takes application artifacts and buildpacks and produces a binary droplet artifact coupled with metadata. This metadata can be anything from hosts and route information to the detected buildpack and the default start command. Essentially, any information that comes out of the entire buildpack compilation and release process can be encapsulated via the buildpack's metadata.

The internals of Diego know nothing about buildpacks. The Cloud Controller, via the Application Life-Cycle Binaries, provides all the required buildpacks to run in a container. The Cloud Controller can be conservative on what it downloads; if you specify a buildpack (e.g., the JBP), only that buildpack will be downloaded. If no buildpack is specified, the Cloud Controller will download all buildpacks. Additionally, individual Cells can cache buildpacks so that they do not need to be repeatedly downloaded from the Cloud Controller. Chapter 5 looks at buildpacks in greater detail.

The steps for staging are as follows:

1. The staging process begins with a `cf push` request from the Cloud Foundry CLI to the Cloud Controller (CAPI). Diego's role in the process occurs when the Cloud Controller instructs Diego (via the CC-Bridge) to stage the application. All Tasks and LRPs are submitted to the CC-Bridge via Cloud Foundry's Cloud Controller. The Cloud Controller begins the staging process by sending a *"stage app bits"* request as a task.
2. The CC-Bridge's Stager picks up and handles the *"stage app bits"* request. The Stager constructs a staging message and forwards it to the BBS. Thus, the Stager represents a transformation function.
3. The first step of the BBS is to store the task information. When the task request is stored in the BBS, the BBS is responsible for validating it. At this stage the task request is only stored; no execution of the task has taken place.
4. Diego is now at a point at which it wants to assign a payload (the Task) to a Cell that is best suited to run it. The BBS determines which of the Brain's Auctioneers is active by looking for the Auctioneer that currently holds the lock record. The BBS communicates to the Auctioneer, directing it to find a Cell to run the Task.

5. The Auctioneer optimally distributes Tasks and LRPs to the cluster of Diego Cells via an auction involving the Cell Reps. The Auctioneer asks all of the Reps what they are currently running. It uses the Reps' responses to make a placement decision. After it selects a Cell, it directs the chosen Cell to run the desired Task. You can configure it so that this auction is done every time you push an application; however, you can also batch auctions for performance to reduce the auction overhead.

   There is additional scope for sharding this auction process by AZ. It is inappropriate to cache the auction results because state is changing all the time; for example, a task might complete or an ActualLRP might crash. To look up which Cells are registered (via their Reps), the Auctioneer communicates with the BBS to get the shared system state. Reps report to BBS directly to inform BBS of their current state. When the Auctioneer is aware of the available Reps, it contacts all Reps directly.

6. The chosen Rep is assigned the Task. After a Task/LRP is assigned to a Cell, that Cell will try to allocate containers based on its internal accounting. Inside the Rep, there is a gateway to Garden (the Executor). The Rep runs Tasks/ActualLRPs by asking its in-process Executor to create a container to run generic action recipes. The Executor creates a Garden container and executes the work encoded in the Task/ActualLRP. This work is encoded as a generic, platform-independent recipe of composable actions (we discussed these in "Composable Actions" on page 49). If the Cell cannot perform the Task, it responds to the Auctioneer, announcing that it was unable to run the requested work. Payloads that are distributed are a batch of Tasks rather than one Task per request. This approach reduces chatter and allows the Cell to attempt all requested tasks and report back any Task it was unable to accomplish.

7. The staging Cell uses the instructions in the buildpack and the staging task to stage the application. It obtains the buildpack through the buildpack Application Life-Cycle Binaries via the file server.

8. Assuming that the task completes successfully, the staging task will result in a droplet, and Garden reports that the container has completed all processes. This information bubbles up through the Rep and the Task is now marked as being in a completed state. When a Task is completed it can call back to the BBS using the callback in the Task request. The callback URL then calls back to the Stager so that the Stager knows that the task is complete (Stagers are stateless, so the callback will return to any Stager). The callback is the process responsible for uploading the metadata from Cell to Stager, and the Stager passes the metadata back to the Cloud Controller.

9. The Cloud Controller also provides information as to where to upload the droplet. The Cell that was responsible for staging the droplet can asynchronously upload the droplet back to the Cloud Controller blobstore. The droplet goes back

to the Cloud Controller blobstore via the Cell's Executor. The Executor uploads the droplet to the file server. The file server asynchronously uploads the blobstore to the Cloud Controller's blobstore, regularly polling to find out when the upload has been completed.

10. Additionally the staging Cell streams the output of the staging process so that the developer can troubleshoot application staging problems. After this staging process has completed successfully, the Cloud Controller subsequently issues a *"run application droplet command"* to Diego to run the staged application.

From a developer's perspective, you issue a command to Cloud Foundry and Cloud Foundry will run your app. However, as discussed at the beginning of this chapter, as work flows through the distributed system, Diego components describe their actions using different levels of abstraction. Internal interactions between the Cloud Controller and Diego's internal components are abstracted away from the developer. They are, however, important for an operator to understand in order to troubleshoot any issues.

Summary

Diego is a distributed system that allows you to run and scale *N* number of applications and tasks in containers across a number of Cells. Here are Diego's major characteristics and attributes:

- It is responsible for running and monitoring OCI-compatible images, standalone applications, and tasks deployed to Cloud Foundry.
- It is responsible for container scheduling and orchestration.
- It is agnostic to both client interaction and runtime implementation.
- It ensures applications remain running by reconciling desired state with actual state through establishing eventual consistency, self-healing, and closed feedback loops.
- It has a generic execution environment made up of composable actions, and a composable backend to allow the support of multiple different Windows- and Linux-based workloads.

When you step back and consider the inherent challenges with any distributed system, the solutions to the consistency and orchestration challenges provided by Diego are extremely elegant. Diego has been designed to make the container runtime subsystem of Cloud Foundry modular and generic. As with all distributed systems, Diego is complex. There are many moving parts and the communication flows between them are not trivial. Complexity is fine, however, if it is well defined within bounded contexts. The Cloud Foundry team has gone to great lengths to design

explicit boundaries for the Diego services and their interaction flows. Each service is free to express its work using its own abstraction, and ultimately this allows for a modular composable plug-and-play system that is easy both to use and operate.

CHAPTER 5
Buildpacks and Docker

Apps[1] that have been deployed to Cloud Foundry run as containerized processes. Cloud Foundry supports running OCI–compatible container images such as Docker as first-class citizens. It also supports running a standalone app artifact (e.g., *.jar* file or Ruby app) deployed "*as is*," containerizing apps on the user's behalf.

Users can deploy a containerized image by using the `cf push` command. This command, along with additional arguments, is used for deploying both standalone apps and OCI-compatible container images.

When deploying just a standalone app artifact, Cloud Foundry stages your pushed app by composing a binary artifact known as a *droplet*. A droplet is an encapsulated version of your app along with all of the required runtime components and app dependencies. Cloud Foundry composes this droplet via its buildpack mechanism. The resulting droplet, combined with a stack (filesystem), is equivalent to a container image. Cloud Foundry runs this droplet in the same way as any other OCI-compatible container image: as an isolated containerized process.

When you push an app, Cloud Foundry automatically detects how to run that app and then invokes the use of Application Life-Cycle Binaries (ALB). The correct set of ALBs will be installed on the Cell (the Diego machine) where the app needs to run. For example, if a Docker image is pushed, the Docker ALBs will be used. If your app requires further compilation via a buildpack, using the buildpack ALBs, Cloud Foundry will additionally detect which buildpack is required to stage and run the app.

The ability to deploy both individual apps and Docker images allows for flexibility. Companies already extensively using Docker images can deploy their existing Docker

1 Apps in this context refers both to Long-Running Processes and tasks. Conceptually, a task is simply a short-lived application with finite characteristics and is guaranteed to run at most once.

images to Cloud Foundry. Companies that want to keep their focus on just their apps can use Cloud Foundry to containerize their app artifacts for them. Although Cloud Foundry supports both approaches, there are trade-offs and benefits to each one. This chapter explores those trade-offs and benefits. It then further explains the nature of buildpacks, including a review of their composition, how they work, and how you can modify them.

Why Buildpacks?

There is an explicit ethos to the buildpack approach. The buildpack model promotes a clean separation of roles between the Platform Operator, who provides the buildpack, and the developer, who produces an app, which then consumes the buildpack in order to run. For example, the Platform Operator defines the language support and the dependency resolution (e.g., which versions of Tomcat or OpenJDK are supported). The developer is then responsible only for providing the app artifacts. This separation of concerns is advantageous to both the developer and the app operator.

Generally speaking, developers should not need to be concerned with building containers or tweaking middleware internals; their focus should be on the business logic of their app. By adopting the use of buildpacks, the developers' focus shifts from building containers and tinkering with middleware to just providing their app artifact. This shift aims to remove the undifferentiated heavy lifting of container construction in order to promote velocity of the app's business code.

Containerizing apps on the developer's behalf also offers additional productivity, security, and operational benefits because you can always build the resulting container image from the same known, vetted, and trusted components. Explicitly, as you move your app artifact between different Cloud Foundry environments (e.g., the app is pushed to different spaces as it progresses through different stages of a CI pipeline), the resulting compiled droplet can and should be staged with the same app, dependencies, and stack. The buildpack configuration facilitates this repeatability. This leaves only the app source code to require additional vulnerability and security scanning on every commit.

An additional feature of buildpacks is increased velocity when patching common vulnerabilities and exposures (CVEs) that affect the app. You can update buildpacks to Cloud Foundry when runtime CVEs emerge. Rather than updating and then redeploying each container image in turn, the Platform Operator simply updates the buildpack or runtime dependency once. Cloud Foundry can then restage and redeploy each app with the latest updates and patches.

Finally, there is a clear benefit to separating the build and run stages. A droplet is built during the compile phase of the buildpack. Building (staging) the droplet is done on a new container that is then destroyed after the droplet is created and uploa-

ded to a blobstore. To run the app, the compiled droplet is then downloaded into a freshly created container. This separation between staging and running an app means that build tools do not end up alongside the executed droplet, further reducing the attack surface of the running app.

Why Docker?

The Docker image format (along with the standard OCI image format) has gained a huge amount of traction in many companies. There are benefits to encapsulating your app and all of its dependencies into a single executable image that can be moved between different environments. An often-cited benefit is that what you run in development is guaranteed to be the same in your production environment because you are not repeatedly rebuilding the container image per environment. With Cloud Foundry, every `cf push` results in a new droplet because you are restaging the application with a buildpack. This means that if your CI pipeline does a `cf push` to a staging environment and then a `cf push` to a production environment, you will have created two different droplets albeit with exactly the same components. A single image removes any doubt that development and production apps could have subtle differences. For example, with the buildpack approach, if your buildpack dependency configuration is too broad, there is a small risk that you can update a dependency during the execution of your pipeline resulting in a subtly different droplet. It is easy to mitigate this by locking down the buildpack dependency scope through configuration. Nonetheless, this is an important consideration to be aware of at this point. In the future, Cloud Foundry will allow droplets to be portable across different environments through download/upload mechanisms.

Another benefit of using Docker is that Docker images work well on desktop computers, allowing for a fast "getting started" experience. For comparison, a full Cloud Foundry installation on a local box (bosh-lite) requires BOSH skills and therefore some upfront investment. However, you can obtain a lightweight, easy-to-use, single-tenant version of Cloud Foundry by using PCFDev.

The trade-off of using Docker images is that more responsibility remains with the developer who constructs the image. Additionally, there are more components requiring a package scan on every new code commit. For example, the opaque nature of Docker images makes detecting and patching CVEs significantly more difficult than containers constructed and managed by the platform.

Whatever approach you decide on, by design, Cloud Foundry natively supports it. This leaves the choice down to what best suits your operational requirements.

App and Dependency Security Scanning

Many companies have already invested heavily in app security scanning. This includes the app and all of its external build and runtime dependencies. Therefore, there are some compelling benefits of using buildpacks to create a droplet for you. From a security and operational standpoint, it is likely to be more secure to keep just the app as the unit of deployment and allow the vetted components of the platform to handle all remaining dependencies.

Buildpacks Explained

Buildpacks are an essential component when deploying app artifacts to Cloud Foundry. Essentially, a buildpack is a directory filesystem that provides the following:

- Detection of an app framework and runtime support
- App compilation (known as staging), including all the required app dependencies
- Application execution

You can locate buildpacks remotely; for example, on GitHub, accessed via any Git URL, as in the case of the Java Buildpack. Buildpacks can also reside natively on Cloud Foundry through a process of being packaged and uploaded for offline use.

You can specify additional buildpack metadata, such as the app name, RAM, service-binding information, and environment variables, on the command line or in an app manifest file.

Using an app manifest provides an easy way to handle change control because you can check manifests into source control. You can see a simple example of an app manifest in the spring-music repository:

```
applications:
- name: spring-music
  memory: 512M
  instances: 1
  random-route: true
  path: build/libs/spring-music.war
```

Buildpacks typically examine the user-provided artifacts (applications along with their manifest and any `CF CLI` arguments) to determine the following:

- Which dependencies should be downloaded
- How apps and runtime dependencies should be configured

Unlike pushing a Docker image, buildpack-built containerized processes undergo a process known as *staging*.

Staging

Buildpacks only make one requirement on the filesystem: it must contain a *bin* directory containing three scripts:

1. Detect
2. Compile
3. Release

Detect, compile, and release are the three life-cycle stages of the buildpack. The three stages are completely independent and it is not possible to pass variables between these scripts.

Collectively, the detect, compile, and release stages are known in Cloud Foundry parlance as staging. Staging happens on a new clean container. The output of staging is a droplet that is uploaded to the Cloud Controller blobstore for later use.

You can write buildpacks in any language. For Ruby-based buildpacks, the buildpack scripts invoke the following piece of Ruby to ensure that Ruby is passed to the environment to run the scripts:

```
#!/usr/bin/env ruby
```

This basically instructs Bash to use the Ruby interpreter when it is executing the script. Alternatively, if you had a buildpack written in Node.js, the script would provide the path to Node.js. For buildpacks written in Bash, you simply invoke the `detect`, `compile`, and `release` scripts in the buildpack's bin directory, as shown here:

```
$ bin/detect <build-dir>
```

The Java Buildpack

Technically, you can make no assumptions about the environment in which the three buildpack scripts will run, and so historically buildpacks were written using Bash. Cloud Foundry ensures that Ruby will be present, thus the Java buildpack (JBP) has deviated from the standard buildpacks to be written in Ruby. There is a difference in philosophy between the JBP and other buildpacks. With other buildpacks such as the Ruby buildpack, the ability to handle things like enterprise proxy servers is handled by setting up a lot of the environment by hand. The JBP is different. It attempts to handle as much of this environment configuration for you via the buildpack components directly.

For the rest of this chapter, we will use the JBP as a great example of how buildpacks work.

Detect

Detect is called only if Cloud Foundry does not know which buildpack to run. Assuming that the user did not specify which buildpack to use at the outset, detect will be the first script to be run. It is invoked when you push an app to Cloud Foundry. Cloud Foundry will iterate through all known buildpacks, based on buildpack ordering, until it finds the first available buildpack that can run the app.

The detect script will not be run if a particular buildpack was specified by the user; for example:

```
$ cf push <my_app> -b my-buildpack
```

Detect is required to return very little. Strictly speaking, detect only needs to return an exit code (either 0 or some other nonzero integer). The JBP, however, returns a list of key–value pairs that describe what it is going to do; for example, use Java Version = Open Jdk JRE 1.8.0_111 and Tomcat = 8.0.38, etc.

System Buildpack Ordering

There are some important considerations when using the detect script. Because the detect script uses the first available buildpack that can run the app, it is important to define the correct ordering of system buildpacks. For example, both the JBP and a TomEE buildpack could run a WAR file. When you use the `cf push` command without explicitly defining the buildpack, the first buildpack in the buildpack list is used.

For this reason, it is best practice to always explicitly define your desired buildpack and to do so using a manifest that you can check into a source-control repository. With that said, because some users might still rely on the detect script, both the Cloud Foundry operator and user should always pay strict attention to buildpack ordering.

Compile

Compile is responsible for all modification tasks that are required prior to execution. Compile takes the pushed app and turns it to a state in which it is ready to run. The `/bin/compile` script can move files around, change file contents, delete artifacts, or do anything else required to get the app into a runnable state. For example, in the case of Java, it downloads Java (Open JDK), Tomcat, JDBC drivers, and so on, and places all of these dependencies in their required location. If compile needs to reconfigure anything, it can reach into your app and, in the example of the JBP, rewrite the

Spring configuration to ensure that everything is properly set up. The compile phase is when you would make any other modifications such as additional app-specific load balancer configurations.

The JBP Compile Phase

The workflow for the JBP is subtly different from other buildpacks. Most other buildpacks will accept app source code. The JBP requires a precompiled app at the code level; for example, a JAR or WAR file.

As discussed in "A Marketplace of On-Demand Services" on page 37, an app can have various services bound to it; for example, database or app monitoring. The compile script downloads any service agents and puts them in the correct directory. Compile also reconfigures the app to ensure that the correct database is configured. If `cf push` specifies that a specific service should be used, but the service is not available, the deployment will fail and the app will not be staged.

The JBP has some additional capabilities; for example, the Tomcat configuration contains some Cloud Foundry–specific values to enable sending logging output to the Loggregator's Doppler component.

Release

The release stage provides the execution command to run the droplet. The release script is part of the droplet and will run as part of staging.

Some buildpacks (such as Java) need to determine how much memory to allocate. The JBP achieves this by using a program that calculates all required settings, such as how big the heap should be. You can run the script at the start of the application as opposed to only during the staging process. This flexibility allows for scaling because the script will be invoked every time a new instance is instantiated. `cf scale` can scale the number of instances and the amount of memory. If you change the amount of RAM, there is no restaging, but the app will be stopped and restarted with the required amount of memory specified, based on the specific memory weightings used.

The rest of the release script sets up variables and the runtime environment. For example, with Java, environment variables such as `JAVA_HOME` and `JAVA_OPTS` (see Java Options Framework) are set, and then finally, the release script will invoke any app server scripts such as *Catalina.sh* to start Tomcat.

Buildpack Structure

The three aforementioned life-cycle stages (detect, compile, release) are echoed by the code. The code sits in the *<buildpack>/lib* directory, all the tests sit in *<buildpack>/spec*, and, in the case of Ruby-based buildpacks, rake is the task executor.

The *<buildpack>/config* directory contains all the configurations. *Components.yml* is the entry point containing a list of all the configurable components.

For example, when looking at the JREs section, we see the following:

```
jres:
  - "JavaBuildpack::Jre::OpenJdkJRE"
# - "JavaBuildpack::Jre::OracleJRE"
# - "JavaBuildpack::Jre::ZuluJRE"
```

The Oracle JRE is disabled because you need a license from Oracle to use it.

Here's the OpenJDK configuration YAML:

```
jre:
  version: 1.8.0_+
  repository_root: "{default.repository.root}/openjdk/{platform}/{architecture}"
memory_calculator:
  version: 1.+
  repository_root: "{default.repository.root}/memory-calculator/{platform}/↵
{architecture}"
  memory_sizes:
    metaspace: 64m..
    permgen: 64m..
  memory_heuristics:
    heap: 75
    metaspace: 10
    permgen: 10
    stack: 5
    native: 10
```

This specifies the use of Java 8 or above ("above" being denoted via the "+") and the repository root where the JRE is located. In addition, it contains the required configuration for the memory calculator app, including the memory weightings.[2]

2 The use of memory weightings for the memory calculator is in the process of being simplified.

Modifying Buildpacks

Cloud Foundry ships with a set of default built-in system buildpacks. To view the current list of built-in system buildpacks, run the `$ cf buildpacks` command via the Cloud Foundry CLI.

If some of the buildpacks require adjustment, in some cases you can override specific configuration settings.

If your app uses a language or framework that the Cloud Foundry system buildpacks do not support, you can write your own buildpack or further customize an existing buildpack. This is a valuable extension point. Operators can, however, choose to disable custom buildpacks in an entire Cloud Foundry deployment if there is a desire for uniformity of supported languages and runtime configuration.

After you have created or customized your new buildpack, you can consume the new buildpack by doing either of the following:

- Specifying the URL of the new repository when pushing Cloud Foundry apps
- Packaging and uploading the new buildpack to Cloud Foundry, making it available alongside the existing system buildpacks

For more information on adding a buildpack to Cloud Foundry, go to the Cloud Foundry documentation page.

Overriding Buildpacks

If you only need to change configuration values such as, in the case of Java, the default version of Java or the Java memory weightings, you can override the buildpack configuration by using environment variables. The name of the overriding environment variable must match the configuration file that you want to override (with the *.yml* extension) and it must be prefixed with `JBP_CONFIG`. The value of the environment variable should be valid inline YAML.

As an example, to change the default version of Java to 7 and adjust the memory heuristics, you can apply the following environment variable to the app:

```
$ cf set-env my-application JBP_CONFIG_OPEN_JDK_JRE '[jre: {version: 1.7.0_+},
+ memory_calculator: {memory_heuristics: {heap: 85, stack: 10}}]'
```

If the key or value contains a special character such as ":", you will need to escape them by using double quotes. Here is an example showing how to change the default repository path for the buildpack:

```
$ cf set-env my-application JBP_CONFIG_REPOSITORY
+ '[ default_repository_root: "http://repo.example.io" ]'
```

You cannot apply a new configuration using this process: you can only override an existing configuration. All new configurations require buildpack modifications, as discussed in "Modifying Buildpacks" on page 91.

The ability to override any configuration in a *config.yml* file has made simple configuration changes to the JBP very straightforward. You can specify environment variables both on the command line or in an app manifest file. You can find more detailed advice on extending the JBP at *https://github.com/cloudfoundry/java-buildpack/blob/master/docs/extending.md*.

Using Custom or Community Buildpacks

It is worth noting that the Cloud Foundry community provides additional external community buildpacks for use with Cloud Foundry.

You can find a complete list of community buildpacks at *https://github.com/cloudfoundry-community/cf-docs-contrib/wiki/Buildpacks*.

Forking Buildpacks

You might have a requirement to extend or modify the buildpack; for example, maybe you need to add an additional custom monitoring agent. The buildpack feature supports modification and extension through the use of the Git repository forking functionality to create a copy of the buildpack repository. This involves making any required changes in your copy of the repository. When forking a buildpack, it is recommended you synchronize subsequent commits from upstream.

Best practice is that if the modifications are generally applicable to the Cloud Foundry community, you should submit the changes back to Cloud Foundry via a pull request.

Restaging

After the first `cf push`, both app files and compiled droplets are retained in the Cloud Controller's blobstore. When you use `cf scale` to scale your app, Cloud Foundry uses the existing droplet.

From time to time, you might want to restage your app in order to recompile your droplet; for example, you might want to pick up a new environment variable or a new app dependency.

The droplet that results from a restage will be completely new. Restage reruns the buildpack against the existing app files (source, JAR, WAR, etc.) stored in the Cloud

Controller blobstore. The restage process picks up all new buildpack updates and any new runtime dependencies that the buildpack can accept. For example, if you, the Platform Operator, have specified the use of Java 8 or above (jre version: 1.8.0_+), the latest available version of Java 8 will be selected. If a specific version of Java was specified (such as jre version: 1.8.0_111), only that version will be selected.

As with `cf push`, `cf restage` runs whatever buildpack is associated with the app. By default, you do not need to specify a buildpack; the platform will run the buildpack *detect* script in the order specified by the command `system buildpacks`. Alternatively, you can explicitly specify a buildpack name or URL. Diego Cells support both *.git* and *.zip* buildpack URLs.

Packaging and Dependencies

There are different approaches that you can take when accessing a buildpack and its dependencies. The approach you choose is determined by two concerns:

- How you access the buildpack
- How you access the buildpack dependencies

You can access the buildpack either remotely, via a Git URL, or by packaging and uploading to Cloud Foundry. You can access buildpack dependencies either by the buildpack remotely (often referred to as online or remote dependencies) or packaged along with a packaged buildpack (referred to as offline dependencies).

Given these considerations, there are three standard approaches to consuming buildpacks:

Online
: You access this via a Git URL with both buildpack and dependencies being pulled from a remote repository.

Minimal-package
: This is a packaged version of the buildpack that is as minimal as possible. The buildpack is uploaded to Cloud Foundry's blobstore, but it is configured to connect to the network for all dependencies. This package is about 50 KB in size.

Offline-package
: This version of the buildpack is designed to run without network access. It packages the latest version of each dependency (as configured in the *config* directory) and disables remote_downloads. This package is about 180 MB in size.

Cloud Foundry deployments residing within an enterprise often have limited access to dependencies due to corporate regulations. Therefore, the second or third options are generally established within an enterprise setting.

With all three approaches, it is recommended that you maintain a local mirror of the buildpack dependencies hosted by Cloud Foundry on Amazon S3. To clone this repository, follow the instructions at the Cloud Foundry GitHub repo. With technologies such as Artifactory, you can set up a pull-through model that watches the source blobstore and pulls down any updates onto your local mirror for internal use only. This approach allows the security team to package-scan all dependencies and provides the Platform Operators with a level of governance over what dependencies can be consumed. It also makes it possible for you to run Cloud Foundry without requiring internet access.

Thus, the decision criterion for these options is one of flexibility versus governance.

The offline-package approach allows for complete control over the buildpack and dependencies. Packing provides an explicit guarantee of known, vetted, and trusted dependencies used to deploy the app. The downside is that you will need an additional CI pipeline (see the section that follows) to build and upload any buildpack and dependency changes.

The advantage of the online approach is that it provides the flexibility to make changes to both the buildpack and the consumption of dependencies without the need to run the changes through a pipeline. You can mitigate concerns surrounding control by strict governance of the mirrored dependency repository. For example, if you want to disable the use of Java 7, you can simply remove it from the repository and update the buildpack accordingly. If a developer then reconfigures his custom buildpack to use Java 7, his deployment will fail. Online buildpacks provide the most flexibility, but without the proper governance, this approach can lead to buildpack sprawl. This governance is managed in a production environment through the use of a pipeline to deploy apps. Buildpack sprawl during development is not a bad thing, provided developers keep in mind the available buildpack options and availability of app dependencies in the production environment.

Buildpack and Dependency Pipelines

When using the JBP, the approach of using a local mirror for dependencies involves forking the buildpack and updating the dependency repository. Keeping this mirror and forked buildpack up-to-date and synchronized with the online buildpack and latest dependencies is vital to ensure that you are guarding against the latest CVEs. To this point, it is prudent to set up a pipeline to maintain buildpack concurrency.

You can set up the dependency pipeline flow as follows:

- Trigger weekly updates from an RSS feed of CVEs that pertain to java_buildpack dependencies (or invoked as soon as a patch is added to the Amazon S3 repository via a pull-down mechanism)

- Use scripts to pull down pertinent items from Cloud Foundry's Amazon S3 buckets (see the Cloud Foundry GitHub repo)
- Push all new dependencies to the local mirror repository, removing any outdated or compromised dependencies

You can set up the buildpack pipeline flow for packaging the java_buildpack as follows:

- Git clone and pull down the latest version of the java_buildpack repository
- Override the dependency repository to point to your local buildpack repository (e.g., an Artifactory repository)
- Push the online buildpack to Cloud Foundry
- Build offline buildpack and push offline buildpack to Cloud Foundry
- Restage all affected apps

Note that even if you are using the online-buildpack approach, it is still valuable to have the offline buildpack available in case there is any downtime of your local repository.

Summary

Cloud Foundry supports pushing Docker images and standalone apps and tasks. Upon pushing an app or task, Cloud Foundry uses a buildpack to containerize and run your app artifact. Buildpacks are a vital component in the deployment chain because they are responsible for transforming deployed code into a droplet, which can then be combined with a stack and executed on a Diego Cell in a container.

Buildpacks enable Cloud Foundry to be truly polyglot and provide an essential extension point to Cloud Foundry users. They also facilitate a separation of concerns between operators who provide the language support and runtime dependencies, and developers who are then free to keep their focus on just their app code.

Whatever approach you choose, be it deploying a standalone app or a Docker image, Cloud Foundry supports both as first-class citizens.

About the Author

Duncan Winn has been working on Cloud Foundry for Pivotal since that company formed in 2013. Currently he works on Pivotal's Platform Architecture Team, helping companies to install and configure hardened Cloud Foundry environments and related services to get the most out of Cloud Foundry in an enterprise setting. Prior to moving to the United States, he was the EMEA Cloud Foundry developer advocate for Pivotal and ran Pivotal's Platform Architecture Team in London. He is actively involved in the Cloud Foundry community blogging, running meetups, and writing books.

Colophon

The animal on the cover of *Cloud Foundry: The Definitive Guide* is a black-winged kite (*Elanus caeruleus*).

It has distinctive long falcon-like wings with white, gray, and black plumage; its eyes are forward-facing with red irises. Its wings extend past its tail when perched, often on roadside wires. To balance itself, it jerks its tail up and down and adjusts its wings. This kite is recognizable for its habit of hovering over open grasslands. The kite doesn't migrate, but makes short distance movements based on weather.

Black-winged kites nest in loose platforms made of twigs. Females spend more energy constructing the nest than males, and lay 3–4 eggs in them during breeding season. Both parents incubate the eggs, but the male spends more time foraging for food, which includes grasshoppers, crickets, other large insects, lizards, and rodents.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Lydekker's Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.