O'REILLY®

Compliments of Pivotal®

Beyond the Twelve-Factor App

Exploring the DNA of Highly Scalable, Resilient Cloud Applications

Kevin Hoffman

Pivotal **Cloud Foundry**®

Cloud Native At Your Service

Trusted by Fortune 100 enterprises including Allstate, Verizon, Daimler AG, GE, Philips, and more, Pivotal Cloud Foundry® is the comprehensive Cloud Native platform for building your future.

Ship new features and applications in days or hours, instead of months

Stop building IT infrastructure, start building your business

Sleep better at night with the platform that keeps your services running

Begin your Cloud Native journey today at **pivotal.io/platform**

Pivotal®

Beyond the Twelve-Factor App

Exploring the DNA of Highly Scalable, Resilient Cloud Applications

Kevin Hoffman

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

Beyond the Twelve-Factor App
by Kevin Hoffman

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: Melanie Yarbrough
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

April 2016: First Edition

Revision History for the First Edition
2016-04-26: First Release

978-1-491-94401-1

[LSI]

Table of Contents

Foreword

Understanding how to design systems to run in the cloud has never been more important than it is today. Cloud computing is rapidly transitioning from a niche technology embraced by startups and tech-forward companies to the foundation upon which enterprise systems build their future. In order to compete in today's marketplace, organizations large and small are embracing cloud architectures and practices.

At Pivotal, my job is to ensure customers succeed with Cloud Foundry. On a typical engagement, I focus mostly on working with operations teams to install and configure the platform, as well as training them to manage, maintain, and monitor it. We deliver a production-grade, fully automated cloud application runtime and hand it over to developers to seize the benefits of the cloud. But how is this achieved? Developers are often left with many questions about the disciplines and practices they should adopt to build applications designed to take advantage of everything the cloud offers. *Beyond the Twelve-Factor App* answers those questions and more.

Whether you are building new applications for the cloud or seeking to migrate existing applications, *Beyond the Twelve-Factor App* is an essential guide that should be on the shelf of every developer and architect targeting the cloud.

— Dan Nemeth, Advisory Solutions Architect, Pivotal

Preface

Buzzwords are the result of our need to build a shared language that allows us to communicate about complex topics without having to stop and do a review. Shared terminology isn't just convenient, it's essential for decision making, architecture design, debates, and even just friendly discussion.

The *Twelve-Factor Application* is one of these phrases that is gaining traction and is being passed around during planning meetings, discussions over coffee, and architecture review sessions.

The problem with shared context and common language like buzzwords is that not everyone has the same understanding. *Twelve-Factor* to one person might mean something entirely different to someone else, and many readers of this book might not have any exposure to the 12 factors.

The goal of this book is to provide detail on what exactly Twelve-Factor applications are so that hopefully everyone who has read the book shares the same understanding of the factors. Additionally, this book aims to take you *beyond* the 12 factors, expanding on the original guidelines to accommodate modern thinking on building applications that don't just function in the cloud, but *thrive*.

The Original 12 Factors

In the early days of the cloud, startups appeared that offered something with which few developers or companies had any experience. It was a new level of abstraction that offered IT professionals freedom from a whole category of nonfunctional requirements. To

some, this was a dark and scary frontier. Others embraced this new frontier as if all of their prayers had been answered.

One of the early pioneers laying claim to territory in the public cloud market was Heroku. It offered to host your application for you, and all you had to do was build your application and push it via git, and then the cloud took over, and your application magically worked online.

Heroku's promise was that you no longer even needed to worry about infrastructure; all you had to do was build your application in a way that took advantage of the cloud, and everything would be just fine.

The problem was that most people simply had no idea how to build applications in a way that was "cloud friendly." As I will discuss throughout this book, cloud-friendly applications don't just run in the cloud; they embrace elastic scalability, ephemeral filesystems, statelessness, and treating everything as a service. Applications built this way can scale and deploy rapidly, allowing their development teams to add new features and react quickly to market changes.

Many of the cloud anti-patterns still being made today will be discussed throughout this book. Early adopters didn't know what you could and could not do with clouds, nor did they know the design and architecture considerations that went into building an application destined for the cloud. This was a new breed of application, one for which few people had a frame of reference.

To solve this problem (and to increase their own platform adoption), a group of people within Heroku developed the *12 Factors* in 2012. This is essentially a manifesto describing the rules and guidelines that needed to be followed to build a *cloud-native*[1] application.

The goal of these 12 factors was to teach developers how to build cloud-ready applications that had *declarative* formats for automation and setup, had a *clean contract* with the underlying operating system, and were dynamically *scalable*.

1 For many people, cloud native and 12 factor are synonymous. One of the goals of this book is to illustrate that there is more to being cloud native than just adhering to the original 12 factors. In Heroku's case, cloud native really meant "works well on Heroku."

These 12 factors were used as guidelines to help steer development of new applications, as well as to create a scorecard with which to measure existing applications and their suitability for the cloud:

Codebase
: One codebase tracked in revision control, many deploys

Dependencies
: Explicitly declare and isolate dependencies

Configuration
: Store configuration in the environment

Backing Services
: Treat backing services as attached resources

Build, release, run
: Strictly separate build and run stages

Processes
: Execute the app as one or more stateless processes

Port binding
: Export services via port binding

Concurrency
: Scale out via the process model

Disposability
: Maximize robustness with fast startup and graceful shutdown

Dev/prod parity
: Keep development, staging, and production as similar as possible

Logs
: Treat logs as event streams

Admin processes
: Run admin/management tasks as one-off processes

These factors serve as an excellent introduction to the discipline of building and deploying applications in the cloud and preparing teams for the rigor necessary to build a production pipeline around elastically scaling applications. However, technology has advanced since their original creation, and in some situations, it is necessary

to elaborate on the initial guidelines as well as add new guidelines designed to meet modern standards for application development.

Beyond the Twelve-Factor Application

In this book, I present a new set of guidelines that builds on the original 12 factors. In some cases, I have changed the order of the factor, indicating a deliberate sense of priority. In other cases, I have added factors such as telemetry, security, and the concept of "API first," that should be considerations for any application that will be running in the cloud. In addition, I may add caveats or exceptions to the original factors that reflect today's best practices.

Taking into account the changes in priority order, definition, and additions, this book describes the following facets of cloud-native applications:

1. One codebase, one application
2. API first
3. Dependency management
4. Design, build, release, and run
5. Configuration, credentials, and code
6. Logs
7. Disposability
8. Backing services
9. Environment parity
10. Administrative processes
11. Port binding
12. Stateless processes
13. Concurrency
14. Telemetry
15. Authentication and authorization

12factor.net provided an excellent starting point, a yardstick to measure applications along an axis of cloud suitability. As you will see throughout the book, these factors often feed each other. Properly following one factor makes it easier to follow another, and so on, throughout a virtuous cycle. Once people get caught up in this cycle, they often wonder how they ever built applications any other way.

Whether you are developing a brand new application without the burden of a single line of legacy code or you are analyzing an enter-

prise portfolio with hundreds of legacy applications, this book will give you the guidance you need to get ready for developing cloud-native applications.

CHAPTER 1
One Codebase, One Application

The first of the original factors, *codebase*, originally stated: "One codebase tracked in revision control, many deploys."

When managing myriad aspects of a development team, the organization of code, artifacts, and other apparent minutia is often considered a minor detail or outright neglected. However, proper application of discipline and organization can mean the difference between a one-month production lead time and a one-day lead time.

Cloud-native applications must always consist of a single codebase that is tracked in a version control system. A codebase is a source code repository or a set of repositories that share a common root.

The single codebase for an application is used to produce any number of immutable releases[1] that are destined for different environments. Following this particular discipline forces teams to analyze the seams of their application and potentially identify monoliths that should be split off into microservices.[2] If you have multiple codebases, then you have a system that needs to be decomposed, not a single application.

1 An immutable release is a build artifact that does not change. As will be discussed throughout the book, this kind of artifact is required for testing, ensuring dev/prod parity, and predicting deployment results.

2 See *Building Microservices* by Sam Newman (O'Reilly) for more guidance on splitting monoliths.

The simplest example of violating this guideline is where your application is actually made of up a dozen or more source code repositories. This makes it nearly impossible to automate the build and deploy phases of your application's life cycle.

Another way this rule is often broken is when there is a main application and a tightly coupled worker (or an en-queuer and de-queuer, etc.) that collaborate on the same units of work. In scenarios like this, there are actually multiple codebases supporting a single application, *even if they share the same source repository root*. This is why I think it is important to note that the concept of a codebase needs to imply a more cohesive unit than just a repository in your version control system.

Conversely, this rule can be broken when one codebase is used to produce multiple applications. For example, a single codebase with multiple launch scripts or even multiple points of execution within a single wrapper module. In the Java world, EAR files are a gateway drug to violating the *one codebase* rule. In the interpreted language world (e.g., Ruby), you might have multiple launch scripts within the same codebase, each performing an entirely different task.

Multiple applications within a single codebase are often a sign that multiple teams are maintaining a single codebase, which can get ugly for a number of reasons. Conway's law states that the organization of a team will eventually be reflected in the architecture of the product that team builds. In other words, dysfunction, poor organization, and lack of discipline among teams usually results in the same dysfunction or lack of discipline in the code.

In situations where you have multiple teams and a single codebase, you may want to take advantage of Conway's law and dedicate smaller teams to individual applications or microservices.

When looking at your application and deciding on opportunities to reorganize the codebase and teams onto smaller products, you may find that one or more of the multiple codebases contributing to your application could be split out and converted into a microservice or API that can be reused by multiple applications.

In other words, *one codebase, one application* does not mean you're not allowed to share code across multiple applications; it just means that the shared code is yet another codebase.

This also doesn't mean that all shared code needs to be a microservice. Rather, you should evaluate whether the shared code should be considered a separately released product that can then be vendored[3] into your application as a dependency.

3 Bundled (or vendored) dependencies and dependency management are discussed in Chapter 3.

CHAPTER 2

API First

This chapter discusses an aspect of modern application development not covered by the original 12 factors. Regardless of the type of application you're developing, chances are if you're developing it for the cloud, then your ultimate goal is to have that application be a participant in an ecosystem of services.

Why API First?

Assume for a moment that you have fully embraced all of the other factors discussed in this book. You are building cloud-native applications, and after code gets checked in to your repository, tests are automatically run, and you have release candidates running in a lab environment within minutes. The world is a beautiful place, and your test environment is populated by rainbows and unicorns.

Now another team in your organization starts building services with which your code interacts. Then, another team sees how much fun you're all having, and they get on board and bring their services. Soon you have multiple teams all building services with horizontal dependencies[1] that are all on a different release cadence.

What can happen if no discipline is applied to this is a nightmare of integration failures. To avoid these integration failures, and to for-

1 A traditional dependency graph looks very hierarchical, where A relies on B, which relies on C. In modern service ecosystems, the graphs are much flatter and often far more complicated.

mally recognize your API as a first-class artifact of the development process, *API first* gives teams the ability to work against each other's public contracts without interfering with internal development processes.

Even if you're not planning on building a service as part of a larger ecosystem, the discipline of starting all of your development at the API level still pays enough dividends to make it worth your time.

Building Services API First

These days, the concept of *mobile first* is gaining a lot of traction. It refers to the notion that from the very beginning of your project, everything you do revolves around the idea that what you are building is a product to be consumed by mobile devices. Similarly, *API first* means that what you are building is an API to be consumed by client applications and services.

As I mentioned at the beginning of this book, cloud native is more than just a list of rules or guidelines. It is a philosophy and, for some of us, a way of life. As such, there are guidelines for cloud native that might not necessarily map to specific physical requirements imposed by the cloud but that are vitally important to the habits of people and organizations building modern applications that will be ready for future changes to the cloud landscape.

Built into every decision you make and every line of code you write is the notion that every functional requirement of your application will be met through the consumption of an API. Even a user interface, be it web or mobile, is really nothing more than a consumer of an API.

By designing your API first, you are able to facilitate discussion with your stakeholders (your internal team, customers, or possibly other teams within your organization who want to consume your API) well before you might have coded yourself past the point of no return. This collaboration then allows you to build user stories, mock your API, and generate documentation that can be used to further socialize the intent and functionality of the service you're building.

All of this can be done to vet (and test!) your direction and plans without investing too much in the plumbing that supports a given API.

These days, you'll find that there are myriad tools and standards to support API-first development. There is a standard format for API specification that uses a markdown-like syntax called API Blueprint. This format is far more human readable than JSON (or WSDL, a relic that belongs in a museum) and can be used by code to generate documentation and even *server mocks*, which are invaluable in testing service ecosystems. Tool suites like Apiary provide things like GitHub integration and server mocks. If someone wants to build a client to your API, all you have to do is give her a link to your application on Apiary, where she can read your API Blueprint, see example code for consuming your service, and even execute requests against a running server mock.

In other words, there is absolutely no excuse for claiming that *API first* is a difficult or unsupported path. This is a pattern that can be applied to noncloud software development, but it is particularly well suited to cloud development in its ability to allow rapid prototyping, support a services ecosystem, and facilitate the automated deployment testing and continuous delivery pipelines that are some of the hallmarks of modern cloud-native application development.

This pattern is an extension of the *contract-first* development pattern, where developers concentrate on building the edges or *seams* of their application first. With the integration points tested continuously via CI servers,[2] teams can work on their own services and still maintain reasonable assurance that everything will work together properly.

API first frees organizations from the waterfall, deliberately engineered system that follows a preplanned orchestration pattern, and allows products to evolve into organic, self-organizing ecosystems that can grow to handle new and unforeseen demands.

If you've built a monolith, or even an ecosystem of monoliths, that all interact in tightly coupled ways, then your ability to adapt to new needs or create new consumers of existing functionality is hindered. On the other hand, if you adopt the mentality that all applications are just backing services (more on those later in the book), and that they should be designed API-first, then your system is free to grow,

2 Continuous integration servers can be used to exercise public APIs and integrations between multiple services. Examples of CI servers include Jenkins, Team City, and Wercker.

adapt to new load and demand, and accommodate new consumers of existing services without having to stop the world to re-architect yet another closed system.

Live, eat, and breathe the API-first[3] lifestyle, and your investment will pay off exponentially.

3 Check out ProgrammableWeb and API First, as well as the documentation at Apiary and API Blueprint, for more details on the API-first lifestyle.

CHAPTER 3
Dependency Management

The second of the original 12 factors, *dependencies*, refers to the management of application dependencies: how, where, and when they are managed.

Reliance on the Mommy Server

In classic enterprise environments, we're used to the concept of the *mommy server*. This is a server that provides everything that our applications need and takes care of their every desire, from satisfying the application's dependencies to providing a server in which to host the app. The inverse of a mommy server, of course, is the embedded, or *bootstrapped*,[1] server, where everything we need to run our application is contained within a single build artifact.

The cloud is a maturation of the classic enterprise model, and as such, our applications need to *grow up* to take advantage of the cloud. Applications can't assume that a server or application container will have everything they need. Instead, apps need to bring their dependencies with them. Migrating to the cloud, maturing your development practices, means weaning your organization off the need for mommy servers.

1 The phrase has dubious origins, but *pulling oneself up by one's own bootstraps* is the leading candidate for the origin of the use of this phrase. In short, bootstrapping involves carrying with you everything you need. Bootstrapping is the exemplar for having no external dependencies.

If you've been building applications in languages or frameworks that don't rely on the container model (Ruby, Go, Java with Spring Boot, etc.), then you're already ahead of the game, and your code remains blissfully unaware of containers or mommy servers.

Modern Dependency Management

Most contemporary programming languages have some facility for managing application dependencies. `Maven` and `Gradle` are two of the most popular tools in the Java world, while `NuGet` is popular for .NET developers, `Bundler` is popular for Ruby, and `godeps` is available for Go programmers. Regardless of the tool, these utilities all provide one set of common functionality: they allow developers to declare dependencies and let the tool be responsible for ensuring that those dependencies are satisfied.

Many of these tools also have the ability to isolate dependencies.[2] This is done by analyzing the declared dependencies and bundling (also called *vendoring*) those dependencies into some sub-structure beneath or within the application artifact itself.

A cloud-native application never relies on implicit existence of system-wide packages. For Java, this means that your applications cannot assume that a container will be managing the classpath on the server. For .NET, this means that your application cannot rely on facilities like the Global Assembly Cache. Ruby developers cannot rely on gems existing in a central location. Regardless of language, your code cannot rely on the pre-existence of dependencies on a deployment target.

Not properly isolating dependencies can cause untold problems. In some of the most common dependency-related problems, you could have a developer working on version *X* of some dependent library on his workstation, but version *X+1* of that library has been installed in a central location in production. This can cause everything from runtime failures all the way up to insidious and difficult to diagnose subtle failures. If left untreated, these types of failures can bring down an entire server or cost a company millions through undiagnosed data corruption.

2 Isolated dependencies are dependencies that reside with, or near, the application that needs them rather than in some central repository or shared location.

Properly managing your application's dependencies is all about the concept of repeatable deployments. Nothing about the runtime into which an application is deployed should be assumed that isn't automated. In an ideal world, the application's container is bundled (or bootstrapped, as some frameworks called it) inside the app's release artifact—or better yet, the application has no container at all.[3]

However, for some enterprises, it just isn't practical (or possible, even) to embed a server or container in the release artifact, so it has to be combined with the release artifact, which, in many cloud environments like Heroku or Cloud Foundry, is handled by something called a *buildpack*.

Applying discipline to dependency management will bring your applications one step closer to being able to thrive in cloud environments.

3 I don't intend to start a religious war over languages. I only assert that the simplest applications are often the easiest to maintain, and containers add a level of complexity that often leads to more effort spent in diagnosis than development.

CHAPTER 4
Design, Build, Release, Run

Factor 5, *build, release, run*, of the original 12 factors, calls for the strict separation of the build and run stages of development. This is excellent advice, and failing to adhere to this guideline can set you up for future difficulties. In addition to the twelve-factor *build, release, run* trio, the discrete *design* step is crucial.

In Figure 4-1, you can see an illustration of the flow from design to run. Note that this is *not* a waterfall diagram: the cycle from design through code and to run is an iterative one and can happen in as small or large a period of time as your team can handle. In cases where teams have a mature CI/CD pipeline, it could take a matter of minutes to go from design to running *in production*.

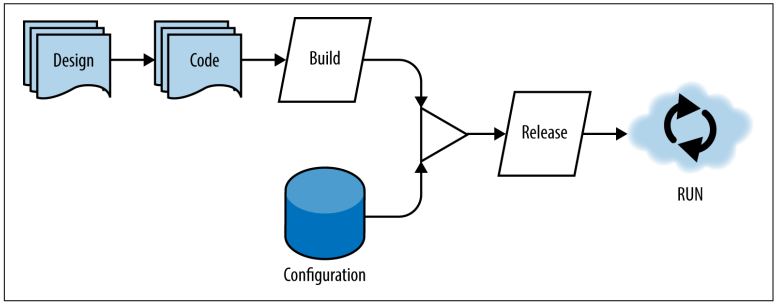

Figure 4-1. The design, build, release, run cycle

A single codebase is taken through the build process to produce a compiled artifact. This artifact is then merged with configuration information that is *external* to the application to produce an *immut-*

able release. The immutable release is then delivered to a cloud environment (development, QA, production, etc.) and run. The key takeaway from this chapter is that each of the following deployment stages is isolated and occurs separately.

Design

In the world of waterfall application development, we spend an inordinate amount of time designing an application before a single line of code is written. This type of software development life cycle is not well suited to the demands of modern applications that need to be released as frequently as possible.

However, this doesn't mean that we don't design at all. Instead, it means we design small features that get released, and we have a high-level design that is used to inform everything we do; but we also know that designs change, and small amounts of design are part of *every iteration* rather than being done entirely up front.

The application developer best understands the application dependencies, and it is during the design phase that arrangements are made to declare dependencies as well as the means by which those dependencies are vendored, or bundled, with the application. In other words, the developer decides what libraries the application is going to use, and how those libraries are eventually going to be bundled into an immutable release.

Build

The build stage is where a code repository is converted into a versioned, binary artifact. It is during this stage that the dependencies declared during the design phase are fetched and bundled into the build artifact (often just simply called a "build"). In the Java world, a build might be a WAR[1] or a JAR file, or it could be a ZIP file or a binary executable for other languages and frameworks.

Builds are ideally created by a Continuous Integration server, and there is a 1:many relationship between builds and deployments. A single build should be able to be released or deployed to any number

1 For a number of reasons, WAR (and EAR) files are looked upon as less cloud native than JAR files, as they imply reliance upon an externally provided server or container.

of environments, and each of those unmodified builds should work as expected. The immutability of this artifact and adherence to the other factors (especially *environment parity*) give you confidence that your app will work in production if it worked in QA.

If you ever find yourself troubleshooting "works on my machine" problems, that is a clear sign that the four stages of this process are likely not as separate as they should be. Forcing your team to use a CI server may often seem like a lot of upfront work, but once running, you'll see that the "one build, many deploys" pattern works.

Once you have confidence that your codebase will work anywhere it should, and you no longer fear production releases, you will start to see some of the truly amazing benefits of adopting the cloud-native philosophy, like continuous deployment and releases that happen *hours* after a checkin rather than months.

Release

In the cloud-native world, the release is typically done by pushing to your cloud environment. The output of the build stage is combined with environment- and app-specific configuration information to produce another immutable artifact, a *release*.

Releases need to be unique, and every release should ideally be tagged with some kind of unique ID, such as a timestamp or an auto-incrementing number. Thinking back to the 1:many relationship between builds and releases, it makes sense that releases should *not* be tagged with the build ID.

Let's say that your CI system has just built your application and labeled that artifact `build-1234`. The CI system might then release that application to the dev, staging, and production environments. The scheme is up to you, but each of those releases should be unique because each one combined the original build with *environment-specific* configuration settings.

If something goes wrong, you want the ability to audit what you have released to a given environment and, if necessary, to roll back to the previous release. This is another key reason for keeping releases both immutable and uniquely identified.

There are a million different types of problems that arise from an organization's inability to reproduce a release as it appeared at one

point in the past. By having separate build and release phases, and storing those artifacts, rollback and historical auditing is a piece of cake.

Run

The run phase is also typically done by the cloud provider (although developers need be able to run applications locally). The details vary among providers, but the general pattern is that your application is placed within some kind of container (Docker, Garden, Warden, etc.), and then a process is started to launch your application.

It's worth noting that ensuring that a developer can run an application locally on her workstation while still allowing it to be deployed to multiple clouds via CD pipeline is often a difficult problem to solve. It is worth solving, however, because developers need to feel unhindered while working on cloud-native applications.

When an application is running, the cloud runtime is then responsible for keeping it alive, monitoring its health, and aggregating its logs, as well as a mountain of other administrative tasks like dynamic scaling and fault tolerance.

Ultimately, the goal of this guidance is to maximize your delivery speed while keeping high confidence through automated testing and deployment. We get some agility and speed benefits out of the box when working on the cloud; but if we follow the guidelines in this chapter, we can squeeze every ounce of speed and agility out of our product release pipeline without sacrificing our confidence in our application's ability to do its job.

CHAPTER 5

Configuration, Credentials, and Code

Factor 3 of the original 12 factors only states that you should store configuration in the environment. I believe the configuration guidance should be more explicit.

Configuration Chemistry

Treat configuration, credentials, and code as *volatile substances* that explode when combined.

That may sound a bit harsh, but failing to follow this rule will likely cause you untold frustration that will only escalate the closer you get to production with your application.

In order to be able to keep configuration separate from code and credentials, we need a very clear definition of configuration. Configuration refers to any value that can vary across deployments (e.g., developer workstation, QA, and production). This could include:

- URLs and other information about backing services, such as web services, and SMTP servers
- Information necessary to locate and connect to databases
- Credentials to third-party services such as Amazon AWS or APIs like Google Maps, Twitter, and Facebook

- Information that might normally be bundled in properties files or configuration XML, or YML

Configuration does *not* include internal information that is part of the application itself. Again, if the value remains the same across all deployments (it is intentionally part of your immutable build artifact), then it isn't configuration.

Credentials are extremely sensitive information and have absolutely no business in a codebase. Oftentimes, developers will extract credentials from the compiled source code and put them in properties files or XML configuration, but this hasn't actually solved the problem. Bundled resources, including XML and properties files, are still part of the codebase. This means credentials bundled in resource files that ship with your application are still violating this rule.

Treat Your Apps Like Open Source

A litmus test to see if you have properly externalized your credentials and configuration is to imagine the consequences of your application's source code being pushed to GitHub.

If the general public were to have access to your code, have you exposed sensitive information about the resources or services on which your application relies? Can people see internal URLs, credentials to backing services, or other information that is either sensitive or irrelevant to people who don't work in your target environments?

If you can open source your codebase without exposing sensitive or environment-specific information, then you've probably done a good job isolating your code, configuration, and credentials.

It should be immediately obvious why we don't want to expose credentials, but the need for external configuration is often not as obvious. External configuration supports our ability to deploy immutable builds to multiple environments *automatically* via CD pipelines and helps us maintain development/production environment parity.

Externalizing Configuration

It's one thing to say that your application's configuration should be *externalized*, but it's a whole different matter to actually do it. If you're working with a Java application, you might be bundling your release artifact with properties files. Other types of applications and languages tend to favor YAML files, while .NET applications traditionally get configuration from XML-based *web.config* and *machine.config* files.

You should consider *all* of these things to be *anti-patterns* for the cloud. All of these situations prevent you from varying configuration across environments while still maintaining your immutable release artifact.

A brute-force method for externalizing your configuration would be to get rid of all of your configuration files and then go back through your codebase and modify it to expect all of those values to be supplied by environment variables. Environment variables are considered the best practice for externalized configuration, especially on cloud platforms like Cloud Foundry or Heroku.

Depending on your cloud provider, you may be able to use its facility for managing *backing services* or *bound services* to expose structured environment variables containing service credentials and URLs to your application in a secure manner.

Another highly recommended option for externalizing configuration is to actually use a server product designed to expose configuration. One such open source server is Spring Cloud Configuration Server, but there are countless other products available. One thing you should look for when shopping for a configuration server product is support for revision control. If you are externalizing your configuration, you should be able to secure data changes as well as obtain a history of who made what changes and when. It is this requirement that makes configuration servers that sit on top of version control repositories like *git* so appealing.

CHAPTER 6
Logs

In this chapter, I discuss the 11th factor, *logs*.

Logs should be treated as *event streams*, that is, logs are a sequence of events emitted from an application in time-ordered sequence. The key point about dealing with logs in a cloud-native fashion is, as the original 12 factors indicate, a truly cloud-native application never concerns itself with routing or storage of its output stream.

Sometimes this concept takes a little bit of getting used to. Application developers, especially those working in large enterprises, are often accustomed to rigidly controlling the shape and destination of their logs. Configuration files or config-related code set up the location on disk where the log files go, log rotation and rollover policies to deal with log file size and countless other minutiae.

Cloud applications can make no assumptions about the file system on which they run, other than the fact that it is ephemeral. A cloud-native application writes all of its log entries to `stdout` and `stderr`. This might scare a lot of people, fearing the loss of control that this implies.

You should consider the aggregation, processing, and storage of logs as a nonfunctional requirement that is satisfied not by your application, but by your cloud provider or some other tool suite running in cooperation with your platform. You can use tools like the **ELK** stack (**E**lasticSearch, **L**ogstash, and **K**ibana), Splunk, Sumologic, or any number of other tools to capture and analyze your log emissions.

Embracing the notion that your application has *less* work to do in the cloud than it does in the enterprise can be a liberating experience.

When your applications are decoupled from the knowledge of log storage, processing, and analysis, your code becomes simpler, and you can rely on industry-standard tools and stacks to deal with logs. Moreover, if you need to change the way in which you store and process logs, you can do so without modifying the application.

One of the many reasons your application should not be controlling the ultimate destiny of its logs is due to elastic scalability. When you have a fixed number of instances on a fixed number of servers, storing logs on disk seems to make sense. However, when your application can dynamically go from 1 running instance to 100, and you have no idea *where* those instances are running, you need your cloud provider to deal with aggregating those logs on your behalf.

Simplifying your application's log emission process allows you to reduce your codebase and focus more on your application's core business value.

CHAPTER 7
Disposability

Disposability is the ninth of the original 12 factors.

On a cloud instance, an application's life is as ephemeral as the infrastructure that supports it. A cloud-native application's processes are disposable, which means they can be started or stopped rapidly. An application cannot scale, deploy, release, or recover rapidly if it cannot start rapidly and shut down gracefully. We need to build applications that not only are aware of this, but also *embrace* it to take full advantage of the platform.

Those used to developing in the enterprise world with creatures like application containers or large web servers may be used to extremely long startup times measured in minutes. Long startup times aren't limited to just legacy or enterprise applications. Software written in interpreted languages or just written poorly can take too long to start.

If you are bringing up an application, and it takes minutes to get into a steady state, in today's world of high traffic, that could mean hundreds or thousands of requests get denied while the application is starting. More importantly, depending on the platform on which your application is deployed, such a slow start-up time might actually trigger alerts or warnings as the application fails its health check. Extremely slow start-up times can even prevent your app from starting at all in the cloud.

If your application is under increasing load, and you need to rapidly bring up more instances to handle that load, any delay during

startup can hinder its ability to handle that load. If the app does not shut down quickly and gracefully, that can also impede the ability to bring it back up again after failure. Inability to shut down quickly enough can also run the risk of failing to dispose of resources, which could corrupt data.

Many applications are written such that they perform a number of long-running activities during startup, such as fetching data to populate a cache or preparing other runtime dependencies. To truly embrace cloud-native architecture, this kind of activity needs to be dealt with separately. For example, you could externalize the cache into a *backing service* so that your application can go up and down rapidly without performing front-loaded operations.

CHAPTER 8
Backing Services

Factor 4 states that you should *treat backing services as bound resources.*

This sounds like good advice, but in order to follow this advice, we need to know what backing services and bound resources are.

A *backing service* is any service on which your application relies for its functionality. This is a fairly broad definition, and its wide scope is intentional. Some of the most common types of backing services include data stores, messaging systems, caching systems, and any number of other types of service, including services that perform line-of-business functionality or security.

When building applications designed to run in a cloud environment where the filesystem must be considered ephemeral, you also need to treat file storage or disk as a backing service. You shouldn't be reading to or writing from files on disk like you might with regular enterprise applications. Instead, file storage should be a backing service that is bound to your application as a resource.

Figure 8-1 illustrates an application, a set of backing services, and the resource bindings (connecting lines) for those services. Again, note that file storage is an abstraction (e.g., Amazon S3) and not direct OS-level access to a disk.

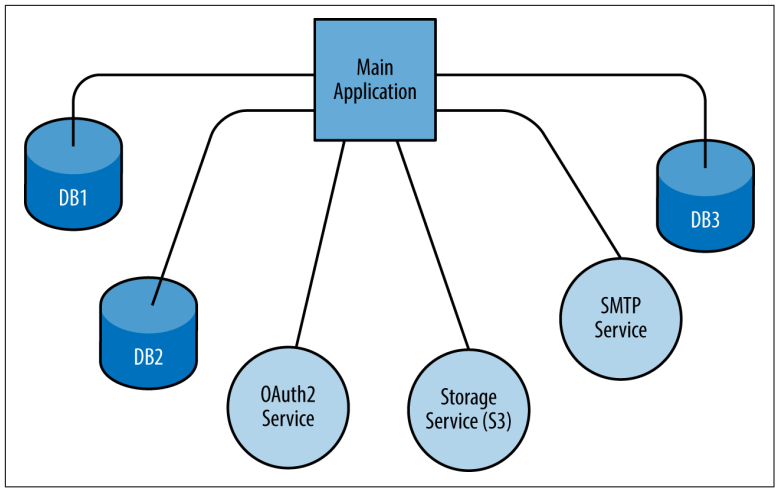

Figure 8-1. An application relying on backing services

A bound resource is really just a means of connecting your application to a backing service. A resource binding for a database might include a username, a password, and a URL that allows your application to consume that resource.

I mentioned earlier in the book that we should have externalized configuration (separated from credentials and code) and that our release products must be immutable. Applying these other rules to the way in which an application consumes backing services, we end up with a few rules for resource binding:

- An application should *declare* its need for a given backing service but allow the cloud environment to perform the actual resource binding.
- The binding of an application to its backing services should be done via external configuration.
- It should be possible to attach and detach backing services from an application at will, *without re-deploying the application*.

As an example, assume that you have an application that needs to communicate with an Oracle database. You code your application such that its reliance on a particular Oracle database is *declared* (the means of this declaration is usually specific to a language or toolset). The source code to the application assumes that the configuration of the resource binding takes place external to the application.

This means that there is *never* a line of code in your application that tightly couples the application to a specific backing service. Likewise, you might also have a backing service for sending email, so you know you will communicate with it via SMTP. But the exact implementation of the mail server should have no impact on your application, nor should your application ever rely on that SMTP server existing at a certain location or with specific credentials.

Finally, one of the biggest advantages to treating backing services as bound resources is that when you develop an application with this in mind, it becomes possible to attach and detach bound resources *at will*.

Let's say one of the databases on which your application relies is not responding. This causes a cascading failure effect and endangers your application. A classic enterprise application would be helpless and at the mercy of the flailing database.

Circuit Breakers

There is a pattern supported by libraries and cloud offerings called the *circuit breaker* that will allow your code to simply stop communicating with misbehaving backing services, providing a fallback or failsafe path. Since a circuit breaker often resides in the binding area between an application and its backing services, you must first embrace backing services before you can take advantage of circuit breakers.

A cloud-native application that has embraced the bound-resource aspect of backing services has options. An administrator who notices that the database is in its death throes can bring up a fresh instance of that database and then *change the binding* of your application to point to this new database.

This kind of flexibility, resilience, and loose coupling with backing services is one of the hallmarks of a truly modern, cloud-native application.

CHAPTER 9
Environment Parity

The tenth of the 12 factors, *dev/prod parity*, instructed us to keep all of our environments as similar as possible.

While some organizations have done more evolving, many of us have likely worked in an environment like this: the shared development environment has a different scaling and reliability profile than QA, which is also different than production. The database drivers used in dev and QA are different than production. Security rules, firewalls, and other environmental configuration settings are also different. Some people have the ability to deploy to some environments, but not others. And finally, the worst part of it all, is people fear deployment, and they have little to no confidence that if the product works in one environment, it will work in another.

When discussing the *design, build, release, run* cycle, I brought up the notion that the "It works on my machine" scenario is a cloud-native anti-pattern. The same is true for other phrases we've all heard right before losing hours or days to firefighting and troubleshooting: "It works in QA" and "It works in prod."

The purpose of applying rigor and discipline to *environment parity* is to give your team and your entire organization the confidence that the application *will work everywhere*.[1]

1 "Everywhere" here deserves some air quotes, or at least an asterisk. By everywhere, I actually mean all of the approved target environments, not all possible environments in all possible locations.

While the opportunities for creating a gap between environments are nearly infinite, the most common culprits are usually:

- Time
- People
- Resources

Time

In many organizations, it could take weeks or months from the time a developer checks in code until the time it reaches production. In organizations like this, you often hear phrases like "the third-quarter release" or "the December 20xx release." Phrases like that are a warning sign to anyone paying attention.

When such a time gap occurs, people often forget what changes went into a release (even if there are adequate release notes), and more importantly, the developers have forgotten what the code looked like.

Adopting a modern approach, organizations should strive to reduce the time gap from check-in to production, taking it from weeks or months to *minutes or hours*. The end of a proper CD pipeline should be the execution of automated tests in different environments until the change is automatically pushed *to production*. With the cloud supporting zero-downtime deployment, this pattern can become the norm.

This idea often scares people, but once developers get into the habit of coding knowing their code will be in production the same day as a check in, discipline and code quality often skyrocket.

People

Historically, the types of people deploying applications were directly related to the size of the company: in smaller companies, developers are usually involved in everything from coding through deployment; whereas in larger organizations, there are more handoffs, and more people and teams involved.

The original 12 factors indicate that the developers and deployers should be the same people, and this makes a lot of sense if your tar-

get is a black-box public cloud like Heroku; but this practice falls down when your target is a private cloud within a large enterprise.

Further, I contend that *humans* should never be deploying applications at all, at least not to any environment other than their own workstations or labs. In the presence of a proper build pipeline, an application will be deployed to all applicable environments automatically and can manually be deployed to other environments based on security restrictions within the CI tool and the target cloud instance.

In fact, even if you are targeting a public cloud provider, it is still possible to use cloud-hosted CD tools like CloudBees or Wercker to automate your testing and deployments.

While there are always exceptions, I contend that if you cannot deploy with a single press of a button, or automatically in response to certain events, then you're doing it wrong.

Resources

When we're sitting at our desks and we need to get something up and running quickly, we all make compromises. The nature of these compromises can leave us with a little bit of technical debt, or it can set us up for catastrophic failure.

One such compromise is often in the way we use and provision *backing services*. Our application might need a database, and we know that in production we'll be hooking it up to an Oracle or a Postgres server, but it's too much of a pain to set that up to be available locally for development, so we'll compromise and use an in-memory database that is *kind of* like the target database.

Every time we make one of these compromises, we increase the gap between our development and production environments; and the wider that gap is, the less predictability we have about the way our application works. As predictability goes down, so does reliability; and if reliability goes down, we lose the ability to have a *continuous* flow from code check-in to production deployment. It adds a sense of brittleness to everything we do; and the worst part is, we usually don't know the consequences of increasing the dev/prod gap until it's too late.

These days, developers have a nearly infinite set of tools at their disposal. There are so few good excuses left for not using the same resource types across environments. Developers can be granted their own instances of databases (this is especially easy if the database is itself a brokered service on a PaaS instance), or if that's not an option, container tools like Docker can help make "prod like" environments more accessible to developer workstations.

As you evaluate every step in your development life cycle while building cloud-native applications, every decision that increases the functional gap between your deployment environments needs to be flagged and questioned, and you need to resist the urge to mitigate this problem by allowing your environments to differ, even if the difference seems insignificant at the time.

Every Commit Is a Candidate for Deployment

It's been discussed at length in this chapter, and you'll see this particular rule mentioned a number of times throughout this book. *Every commit is a candidate for deployment*.

When building applications in a cloud-first way, every time you commit a change, that change should end up in production after some *short* period of time: basically the amount of time it takes to run all tests, vet that change against all integration suites, and deploy to pre-production environments.

If your development, testing, and production environments differ, even in ways you might think don't matter, then you lose the ability to accurately predict how your code change is going to behave in production. This confidence in the code heading to production is essential for the kind of continuous delivery, rapid deployment that allows applications and their development teams to thrive in the cloud.

CHAPTER 10
Administrative Processes

The twelfth and final factor originally stated, "Run admin/management tasks as one-off processes." I feel that this factor can be misleading. There is nothing inherently wrong with the notion of an administrative process, but there are a number of reasons why you should not use them.

The problem with the original twelve-factor recommendation is that it was written in an opinionated way with a bias toward interpreted languages like Ruby that support and encourage an interactive programming shell.[1] Administrative processes were a feature the authors wanted customers to utilize.

I contend that, in certain situations, the use of administrative processes is actually a *bad* idea, and you should always be asking yourself whether an administrative process is what you want, or whether a different design or architecture would suit your needs better. Examples of administrative processes that should probably be refactored into something else include:

- Database migrations
- Interactive programming consoles (REPLs)
- Running timed scripts, such as a nightly batch job or hourly import
- Running a one-off job that executes custom code only once

1 Such shells are referred to as REPLs, which is an acronym for read-eval-print loop.

First, let's take a look at the issue of timers (usually managed with applications like Autosys or Cron). One thought might be to just internalize the timer and have your application wake itself up every *n* hours to perform its batch operations. On the surface, this looks like a good fix, but what happens when there are 20 instances of your application running in one availability zone, and another 15 running in the other zone? If they're all performing the same batch operation on a timer, you're basically inciting chaos at this point, and corrupt or duplicate data is going to be just one of the many terrible things that arise from this pattern.

Interactive shells are also problematic for a number of reasons, but the largest of those is that even if it were possible to reach that shell, you'd only be interacting with the temporary memory of a single instance. If the application had been built properly as a *stateless process*, then I would argue that there is little to no value in exposing a REPL for in-process introspection.[2]

Next, let's take a look at the mechanics of triggering a timed or batch administrative process. This usually happens with the execution of a shell script by some external timer stimulus like `cron` or `Autosys`. In the cloud, you can't count on being able to invoke these commands, so you need to find some other way to trigger ad hoc activity in your application.

In Figure 10-1, you can see a classic enterprise architecture of an application that has its regular duties and supports batch or timed operation via the execution of shell scripts. This is clearly not going to work in the cloud.

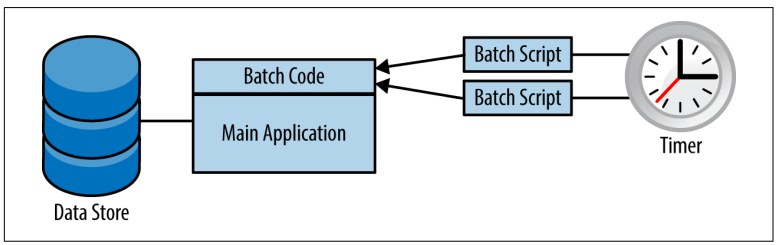

Figure 10-1. Classic enterprise app with batch components

2 Another chapter, Telemetry, actually covers more aspects of application monitoring that even further negate the need for interactive access to a cloud process.

There are several solutions to this problem, but the one that I have found to be most appealing, especially when migrating the rest of the application to be cloud native, is to expose a RESTful endpoint that can be used to invoke ad hoc functionality, as shown in Figure 10-2.

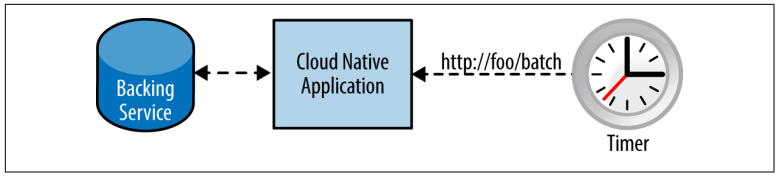

Figure 10-2. App refactored to expose REST endpoint for ad hoc functionality

Another alternative might be to extract the batch-related code from the main application and create a separate microservice, which would also resemble the architecture in the preceding diagram.

This still allows at-will invocation of timed functionality, but it moves the stimulus for this action *outside* the application. Moreover, this method also solves the *at most once* execution problem that you would have from internal timers on dynamically scaled instances. Your batch operation is handled once, by one of your application instances, and you might then interact with other *backing services* to complete the task. It should also be fairly straightforward to secure the batch endpoint so that it can only be operated by authorized personnel. Even more useful is that your batch operation can now scale elastically and take advantage of all the other cloud benefits.

Even with the preceding solution, there are several application architecture options that might make it completely unnecessary to even expose batch or ad hoc functionality within your application.

If you still feel you need to make use of administrative processes, then you should make sure you're doing so in a way that is in line with the features offered by your cloud provider. In other words, don't use your favorite programming language to spawn a new process to run your job; use something designed to run one-off tasks in a cloud-native manner. In a situation like this, you could use a solution like Amazon Web Services Lambdas, which are functions that get invoked on-demand and do not require you to leave provisioned servers up and running like you would in the preceding microservice example.

When you look at your applications, whether they are green field or brown field, just make sure you ask yourself if you really need administrative processes, or if a simple change in architecture could obviate them.

CHAPTER 11
Port Binding

Factor 7 states that cloud-native applications *export services via port binding*.

Avoiding Container-Determined Ports

Web applications, especially those already running within an enterprise, are often executed within some kind of server container. The Java world is full of containers like Tomcat, JBoss, Liberty, and WebSphere. Other web applications might run inside other containers, like Microsoft Internet Information Server (IIS).

In a noncloud environment, web applications are deployed to these containers, and the container is then responsible for assigning ports for applications when they start up.

One extremely common pattern in an enterprise that manages its own web servers is to host a number of applications in the same container, separating applications by port number (or URL hierarchy) and then using DNS to provide a user-friendly facade around that server. For example, you might have a (virtual or physical) host called `appserver`, and a number of apps that have been assigned ports 8080 through 8090. Rather than making users remember port numbers, DNS is used to associate a host name like `app1` with `appserver:8080`, `app2` with `appserver:8081`, and so on.

Avoiding Micromanaging Port Assignments

Embracing platform-as-a-service here allows developers and devops alike to not have to perform this kind of micromanagement anymore. Your cloud provider should be managing the port assignment for you because it is likely also managing routing, scaling, high availability, and fault tolerance, all of which require the cloud provider to manage certain aspects of the network, including routing host names to ports and mapping external port numbers to container-internal ports.

The reason the original 12 factor for *port binding* used the word *export* is because it is assumed that a cloud-native application is *self-contained* and is never injected into any kind of external application server or container.

Practicality and the nature of existing enterprise applications may make it difficult or impossible to build applications this way. As a result, a slightly less restrictive guideline is that *there must always be a 1:1 correlation between application and application server*. In other words, your cloud provider might support a web app container, but it is extremely unlikely that it will support hosting multiple applications within the same container, as that makes durability, scalability, and resilience nearly impossible.

The developer impact of port binding for modern applications is fairly straightforward: your application might run as `http://localhost:12001` when on the developer's workstation, and in QA it might run as `http://192.168.1.10:2000`, and in production as `http://app.company.com`. An application developed with exported port binding in mind supports this environment-specific port binding without having to change any code.

Applications are Backing Services

Finally, an application developed to allow externalized, runtime port binding can act as a backing service for another application. This type of flexibility, coupled with all the other benefits of running on a cloud, is extremely powerful.

CHAPTER 12
Stateless Processes

Factor 6, *processes*, discusses the stateless nature of the processes supporting cloud-native applications.

Applications should execute as a *single, stateless* process. As mentioned earlier in the book, I have a strong opinion about the use of administrative and secondary processes, and modern cloud-native applications should each consist of a *single*,[1] stateless process.

This slightly contradicts the original 12 factor discussion of stateless processes, which is more relaxed in its requirement, allowing for applications to consist of multiple processes.

A Practical Definition of Stateless

One question that I field on a regular basis stems from confusion around the concept of statelessness. People wonder how they can build a process that maintains no state. After all, every application needs *some* kind of state, right? Even the simplest of application leaves some bit of data floating around, so how can you ever have a truly stateless process?

A stateless application makes no assumptions about the contents of memory prior to handling a request, nor does it make assumptions about memory contents after handling that request. The application

1 "Single" in this case refers to a single conceptual process. Some servers and frameworks might actually require more than one process to support your application.

can create and consume transient state in the middle of handling a request or processing a transaction, but that data should all be gone by the time the client has been given a response.

To put it as simply as possible, *all long-lasting state must be external to the application, provided by* ***backing services***. So the concept isn't that state cannot exist; it is that it cannot be maintained within your application.

As an example, a microservice that exposes functionality for user management must be stateless, so the list of all users is maintained in a backing service (an Oracle or MongoDB database, for instance). For obvious reasons, it would make no sense for a database to be stateless.

The Share-Nothing Pattern

Processes often communicate with each other by sharing common resources. Even without considering the move to the cloud, there are a number of benefits to be gained from adopting the *share-nothing* pattern.

Firstly, anything shared among processes is a liability that makes all of those processes more brittle. In many high-availability patterns, processes will share data through a wide variety of techniques to elect cluster leaders, to decide on whether a process is a primary or backup, and so on.

All of these options need to be avoided when running in the cloud. Your processes can vanish at a moment's notice with no warning, and *that's a good thing*. Processes come and go, scale horizontally and vertically, and are highly disposable. This means that anything shared among processes could also vanish, potentially causing a cascading failure.

It should go without saying, but *the filesystem is not a backing service*. This means that you cannot consider files a means by which applications can share data. Disks in the cloud are ephemeral and, in some cases, even read-only.

If processes need to share data, like session state for a group of processes forming a web farm, then that session state should be externalized and made available through a true backing service.

Data Caching

A common pattern, especially among long-running, container-based web applications, is to cache frequently used data during process startup. This book has already mentioned that processes need to start and stop quickly, and taking a long time to fill an in-memory cache violates this principle.

Worse, storing an in-memory cache that your application thinks is always available can bloat your application, making each of your instances (which should be elastically scalable) take up far more RAM than is necessary.

There are dozens of third-party caching products, including Gemfire and Redis, and all of them are designed to act as a backing service cache for your applications. They can be used for session state, but they can also be used to cache data your processes may need during startup and to avoid tightly coupled data sharing among processes.

CHAPTER 13
Concurrency

Factor 8, *concurrency*, advises us that cloud-native applications should *scale out using the process model*. There was a time when, if an application reached the limit of its capacity, the solution was to increase its size. If an application could only handle some number of requests per minute, then the preferred solution was to simply make the application *bigger*.

Adding CPUs, RAM, and other resources (virtual or physical) to a single monolithic application is called *vertical scaling*, and this type of behavior is typically frowned upon in civilized society these days.

A much more modern approach, one ideal for the kind of elastic scalability that the cloud supports, is to scale *out*, or *horizontally*. Rather than making a single big process even larger, you create multiple processes, and then distribute the load of your application among those processes.

Most cloud providers have perfected this capability to the point where you can even configure rules that will dynamically scale the number of instances of your application based on load or other runtime telemetry available in a system.

If you are building disposable, stateless, share-nothing processes then you will be well positioned to take full advantage of horizontal scaling and running multiple, concurrent instances of your application so that it can truly thrive in the cloud.

CHAPTER 14
Telemetry

The concept of telemetry is not among the original 12 factors. Telemetry's dictionary definition implies the use of special equipment to take specific measurements of something and then to transmit those measurements elsewhere using radio. There is a connotation here of remoteness, distance, and intangibility to the source of the telemetry.

While I recommend using something a little more modern than radio, the use of *telemetry* should be an essential part of any cloud-native application.

Building applications on your workstation affords you luxuries you might not have in the cloud. You can inspect the inside of your application, execute a debugger, and perform hundreds of other tasks that give you visibility deep within your app and its behavior.

You don't have this kind of direct access with a cloud application. Your app instance might move from the east coast of the United States to the west coast with little or no warning. You could start with one instance of your app, and a few minutes later, you might have hundreds of copies of your application running. These are all incredibly powerful, useful features, but they present an unfamiliar pattern for real-time application monitoring and telemetry.

NOTE

Treat Apps Like Space Probes

I like to think of pushing applications to the cloud as launching a scientific instrument into space.

If your creation is thousands of miles away, and you can't physically touch it or bang it with a hammer to coerce it into behaving, what kind of telemetry would you want? What kind of data and remote controls would you need in order to feel comfortable letting your creation float freely in space?

When it comes to monitoring your application, there are generally a few different categories of data:

- Application performance monitoring (APM)
- Domain-specific telemetry
- Health and system logs

The first of these, APM, consists of a stream of events that can be used by tools outside the cloud to keep tabs on how well your application is performing. This is something that you are responsible for, as the definition and watermarks of performance are specific to your application and standards. The data used to supply APM dashboards is usually fairly generic and can come from multiple applications across multiple lines of business.

The second, domain-specific telemetry, is also up to you. This refers to the stream of events and data that makes sense to your business that you can use for your own analytics and reporting. This type of event stream is often fed into a "big data" system for warehousing, analysis, and forecasting.

The difference between APM and domain-specific telemetry may not be immediately obvious. Think of it this way: APM might provide you the average number of HTTP requests per second an application is processing, while domain-specific telemetry might tell you the number of widgets sold to people on iPads within the last 20 minutes.

Finally, health and system logs are something that should be provided by your cloud provider. They make up a stream of events, such as application start, shutdown, scaling, web request tracing, and the results of periodic health checks.

The cloud makes many things easy, but monitoring and telemetry are still difficult, probably even *more* difficult than traditional, enterprise application monitoring. When you are staring down the firehose at a stream that contains regular health checks, request audits, business-level events, and tracking data, and performance metrics, that is an incredible amount of data.

When planning your monitoring strategy, you need to take into account how much information you'll be aggregating, the rate at which it comes in, and how much of it you're going to store. If your application dynamically scales from 1 instance to 100, that can also result in a hundredfold increase in your log traffic.

Auditing and monitoring cloud applications are often overlooked but are perhaps some of the most important things to plan and do properly for production deployments. If you wouldn't blindly launch a satellite into orbit with no way to monitor it, you shouldn't do the same to your cloud application.

Getting telemetry done right can mean the difference between success and failure in the cloud.

CHAPTER 15

Authentication and Authorization

There is no discussion of security, authentication, or authorization in the original 12 factors.

Security is a vital part of any application and cloud environment. *Security should never be an afterthought*.

All too often, we are so focused on getting the functional requirements of an application out the door that we neglect one of the most important aspects of delivering any application, regardless of whether that app is destined for an enterprise, a mobile device, or the cloud.

A cloud-native application is a *secure* application. Your code, whether compiled or raw, is transported across many data centers, executed within multiple containers, and accessed by countless clients—some legitimate, most nefarious.

Even if the only reason you implement security in your application is so you have an audit trail of which user made which data change, that alone is benefit enough to justify the relatively small amount of time and effort it takes to secure your application's endpoints.

In an ideal world, all cloud-native applications would secure all of their endpoints with RBAC (role-based access control).[1] Every request for an application's resources should know *who* is making the request, and the roles to which that consumer belongs. These

1 Wikipedia has more information on RBAC, including the NIST RBAC model.

roles dictate whether the calling client has sufficient permission for the application to honor the request.

With tools like OAuth2, OpenID Connect, various SSO servers and standards, as well as a near infinite supply of language-specific authentication and authorization libraries, security should be something that is baked into the application's development from day one, and not added as a bolt-on project after an application is running in production.

CHAPTER 16
A Word on Cloud Native

Now that you have read through a discussion that goes beyond the twelve-factor application and have learned that people often use "12 factor" and "cloud native" interchangeably, it is worth taking a moment for a discussion on the term *cloud native*.

What Is Cloud Native?

Buzzwords and phrases like "SOA," "cloud native," and "microservices" all start because we need a faster, more efficient way to communicate our thoughts on a subject. This is essential to facilitating meaningful conversations on complex topics, and we end up building a *shared context* or a *common language*.

The problem with these buzzwords is that they rely on mutual or common understanding between multiple parties. Like the classic game of telephone[1] on an epic scale, this alleged shared understanding rapidly deteriorates into mutual confusion.

We saw this with SOA (service-oriented architecture), and we're seeing it again with the concept of cloud native. It seems as though every time this concept is shared, the meaning changes until we have as many opinions about cloud native as we do IT professionals.

1 Communication and the development of shared context is a rich subject about which many books have been written.

To understand "cloud native," we must first understand "cloud." Many people assume that "cloud" is synonymous with public, unfettered exposure to the Internet. While there are some cloud offerings of this variety, that's far from a complete definition.

In the context of this book, cloud refers to *Platform as a Service*. PaaS providers expose a platform that hides infrastructure details from the application developer, where that platform resides on top of Infrastructure as a Service (IaaS). Examples of PaaS providers include Google App Engine, Redhat Open Shift, Pivotal Cloud Foundry, Heroku, AppHarbor, and Amazon AWS.

The key takeaway is that cloud is not necessarily synonymous with public, and enterprises are setting up their own private clouds in their data centers, on top of their own IaaS, or on top of third-party IaaS providers like VMware or Citrix.

Next, I take issue with the word "native" in the phrase "cloud native." This creates the mistaken impression that only brand-new, green field applications developed natively within a cloud can be considered cloud native. This is wholly untrue, but since the "cloud native" phrase is now ubiquitous and has seen rapid proliferation throughout most IT circles, I can't use phrases like "cloud friendly," "cloud ready," or "cloud optimized" because they're neither as catchy nor as widely recognized as the original phrase that has now made its way into our vernacular. The following is what I've come up with as a simple definition for a cloud-native application to be:

> A cloud-native application is an application that has been designed and implemented to run on a Platform-as-a-Service installation and to embrace horizontal elastic scaling.

The struggle with adding any more detail is you then start to tread on other people's perspective of what constitutes cloud native, and you potentially run afoul of the "pragmatism versus purism" argument (discussed later in this chapter).

Why Cloud Native?

Not too long ago, it would have been considered the norm to build applications knowing they would be deployed on physical servers—anything from big towers in an air-conditioned room to slim *1U* devices installed in a real data center.

Bare metal deployment was fraught with problems and risk: we couldn't dynamically scale applications, the deployment process was difficult, changes in hardware could cause application failures, and hardware failure often caused massive data loss and significant downtime.

This led to the virtualization revolution. Everyone agreed that bare metal was no longer the way to go, and thus the *hypervisor* was born. The industry decided to put a layer of abstraction on top of the hardware so that we could make deployment easier, to scale our applications horizontally, and to hopefully prevent large amounts of downtime and susceptibility to hardware failure.

In today's always-connected world of smart devices and even smarter software, you have to look long and hard to find a company that doesn't have some kind of software development process as its keystone. Even traditional manufacturing industries, where companies make hard, *physical* things, manufacturing doesn't happen without software. People can't be organized to build things efficiently and at scale without software, and you certainly cannot participate in a global marketplace without it.

Regardless of what industry you're in, you cannot compete in today's marketplace without the ability to rapidly deliver software that simply *does not fail*. It needs to be able to dynamically scale to deal with volumes of data previously unheard of. If you can't handle *big data*, your competitors will. If you can't produce software that can handle massive load, remain responsive, and change as rapidly as the market, your competitors will find a way to do it.

This brings us to the essence of *cloud native*. Gone are the days where companies could get away with being diverted by spending inordinate amounts of time and resources on DevOps tasks, on building and maintaining brittle infrastructures, and fearing the consequences of production deployments that only happen once every blue moon.

Today, we need to be able to focus squarely on the one thing that we do better than our competitors and let platforms take care of our nonfunctional requirements. In his book *Good to Great* (HarperBusiness), Jim Collins asks the question: *are you a hedgehog, or are you a fox?*

The Greek poet and mercenary Archilochus actually first discussed this concept by saying, "The fox knows many things, but the hedgehog knows one big thing." The core of this quote forces us to look at where we spend our time and resources and compare that to the *one big thing* that we want to do. What does your company or team want to accomplish? Chances are, you didn't answer this question with things like failover, resiliency, elastic scalability, or automated deployment. No, what you want to build is the thing that distinguishes you from all the others. You want to build the thing that is the key to your business, and leave all the other stuff to someone (or some*thing*) else.

This is the age of the cloud, and we need to build our applications in a way that embraces this. We need to build our applications so that we can spend the majority of our time working on the hedgehog (the one big thing) and let someone or something else take care of the fox's many small things. Super fast time to market is no longer a nice-to-have; it's a necessity to avoid being left behind by our competition. We want to be able to devote our resources to our business domain, and let other experts deal with the things they do better than us.

By embracing *cloud-native architecture*, and building our applications on the assumption that *everything is a service* and that they will be deployed in a cloud environment, we can get all of these benefits and much more. The question isn't *why cloud-native?* The question you have to ask yourself is why are you *not* embracing cloud native?

The Purist vs the Pragmatist

With all patterns from SOA to REST and everything in between, there are the shining ideals held atop ivory towers, and then there is the reality of real-world development teams, budgets, and constraints. The trick is in determining which ideals on which you will not budge, and which ideals you will allow to get a little muddy in service of the pragmatic needs to get products shipped on time.

Throughout this book, I have mentioned where compromises against the ideal are possible or even common, and was also clear where experience shows we simply cannot acquiesce. The decision is ultimately yours, and we would all be extremely happy if every application we created was a *pure* cloud-native application that never violated a single guideline from this book, but reality and

experience shows that compromise on purist ideals is as ever-present as death and taxes.

Rather than adopting an all-or-nothing approach, learning where and when to compromise on the guidelines in this book is probably the single most important skill to have when planning and implementing cloud-native applications.

CHAPTER 17
Summary

Twelve-factor applications are an excellent start toward building applications that operate in the cloud, but to build cloud-native applications that truly thrive in the cloud, you need to look *beyond* the 12 factors.

My challenge to you is this: evaluate your existing applications against the guidelines set forth in this book and start planning what it would take to get them to run in a cloud. All other benefits aside, eventually, everything is going to be cloud-based the way everything today runs on virtualization.

When you're building a new application, force a decision as to why you should *not* build your application in a cloud-native way.

Embrace continuous integration, continuous delivery, and the production of applications designed to thrive in the cloud, and you will reap rewards far and above just what you get from a cloud-native world.

About the Author

Kevin Hoffman is an Advisory Solutions Architect for Pivotal Cloud Foundry where he helps teach organizations how to build cloud native apps, migrate applications to the cloud, and embrace all things cloud and microservice. He has written applications for just about every type of industry, including autopilot software for quadcopters, waste management, financial services, and biometric security.

In his spare time, when not coding, tinkering, or learning new tech, he also writes fantasy and science fiction books.