**Chapter 2. Running cloud-native applications in production**

*This chapter covers*

* Recognizing why developers should care about operations
* Understanding obstacles to successful deployments
* Eliminating those obstacles
* Implementing continuous delivery
* Impact of cloud-native architectural patterns on operations

As a developer, you want nothing more than to create software that users will love and that will provide them value. When users want more, or you have an idea for something you’d like to bring to them, you’d like to build it and deliver it with ease. And you want your software to run well in production, to always be available and responsive.

Unfortunately, for most organizations, the process of getting software deployed in production is challenging. Processes designed to reduce risk and improve efficiency have the inadvertent effect of doing exactly the opposite, because they’re slow and cumbersome to use. And after the software is deployed, keeping it up and running is equally difficult. The resulting instability causes production-support personnel to be in a perpetual state of firefighting.

Even given a body of well-written, completed software, it’s still difficult to

* Get that software deployed
* Keep it up and running

As a developer, you might think that this is someone else’s problem. Your job is to produce that well-written piece of code; it’s someone else’s job to get it deployed and to support it in production. But responsibility for today’s fragile production environment doesn’t lie with any particular group or individual; instead, the “blame” rests with a system that has emerged from a set of organizational and operational practices that are all but ubiquitous across the industry. The way that teams are defined and assigned responsibility, the way that individual teams communicate, and even the way that software is architected all contribute to a system that, frankly, is failing the industry.

The solution is to design a new system that doesn’t treat production operations as an independent entity, but rather connects software development practices and architectural patterns to the activities of deploying and managing software in production.

In designing a new system, it behooves you to first understand what is causing the greatest pains in the current one. After you’ve analyzed the obstacles you currently face, you can construct a new system that not only avoids the challenges, but also thrives by capitalizing on new capabilities offered in the cloud. This is a discussion that addresses the processes and practices of the entire software delivery lifecycle, from development through production. As a software developer, you play an important role in making it easier to deploy and manage software in production.

**2.1. The obstacles**

No question—handling production operations is a difficult and often thankless job. Working hours usually include late nights and weekends, either when software releases are scheduled or when unexpected outages happen. It isn’t unusual for a fair bit of conflict to arise between application development groups and operations teams, with each blaming the other for failure to adequately serve consumers with superior digital experiences.

But as I said, that isn’t the fault of the ops team nor of the app-dev team. The challenges come from a system that inadvertently erects a series of barriers to success. Although every challenging situation is unique, with a variety of detailed root causes playing a part, several themes are common across almost all organizations. They’re shown in [figure 2.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig01) and are summarized as follows:

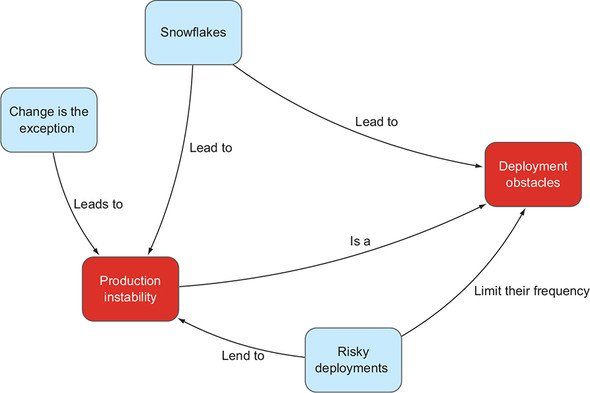
* ***Snowflakes—***Variability across the software development lifecycle (SDLC) contributes to trouble with initial deployments as well as to a lack of stability after the apps are running. Inconsistencies in both the software artifacts being deployed and the environments being deployed to are the problem.
* ***Risky deployments—***The landscape in which software is deployed today is highly complex, with many tightly coupled, interrelated components. As such, a great risk exists that a deployment bringing a change in one part of that complex network will cause rippling effects in any number of other parts of the system. And fear of the consequences of a deployment has the downstream effect of limiting the frequency with which you can deploy.
* ***Change is the exception—***Over the last several decades, we generally wrote and operated software with the expectation that the system where it ran would be stable. This philosophy was probably always suspect. But now, with IT systems being complex and highly distributed, this expectation of infrastructure stability is a complete fallacy.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fn1)**]** As a result, any instability in the infrastructure propagates up into the running application, making it hard to keep running.

***1***

*Wikipedia’s “Fallacies of Distributed Computing” entry (*[*http://mng.bz/pgqw*](http://mng.bz/pgqw)*) provides more details.*

* ***Production instability—***And finally, because deploying into an unstable environment is usually inviting more trouble, the frequency of production deployments is limited.

**Figure 2.1. Factors that contribute to the difficulty in deploying software and keeping it running well in production**



Let’s explore each of these factors further.

**2.1.1. Snowflakes**

“It works on my machine” is a common refrain when the ops team is struggling to stand up an application in production and reaches out to the development team for help. I’ve spoken with professionals at dozens of large enterprises who’ve told of six-, eight-, or even ten-week delays between the time that software is ready for release and the time it’s available to the user. One of the primary reasons for this delay is variability across the SDLC. This variability occurs along two lines:

* A difference in environments
* A difference in the artifacts being deployed

Without a mechanism for providing exactly the same environment from development through testing, staging, and production, it’s easy for software running in one environment to inadvertently depend on something that’s lacking or different in another one. One obvious example of this occurs when differences exist in the packages that the deployed software depends on. A developer might be strict about constantly updating all versions of the Spring Framework, for example, even to the point of automating installs as a part of their build scripts. The servers in the production environment are far more controlled, and updates to the Spring Framework occur quarterly and only after a thorough audit. When the new software lands on that system, tests no longer pass, and resolution likely requires going all the way back to the developer to have them use the production-approved dependencies.

But it isn’t only differences in environment that slow deployments. All too often the artifact being deployed also varies through the SDLC—even when environment-specific values aren’t hardcoded into the implementation (which none of us would ever do, right?). Property files often contain configurations that are directly compiled into the deployable artifact. For example, the JAR file for your Java application includes an application.properties file and, if certain configuration settings are made directly in that file—ones that vary across dev, test, and prod—the JAR files must be different for dev, test, and prod too. In theory, the only differences between each of those JAR files are the contents of the property files, but any recompiling or repackaging of the deployable artifact can, and often does, end up inadvertently bringing in other differences as well.

These snowflakes don’t only have a negative impact on the timeline for the initial deployment; they also contribute greatly to operational instability. For example, let’s say you have an app that has been running in production with roughly 50,000 concurrent users. Although that number doesn’t generally fluctuate too much, you want room for growth. In the user acceptance testing (UAT) phase, you exercise a load with twice that volume, and all tests pass. You deploy the app into production, and all is well for some time. Then, on Saturday morning at 2 A.M., you see a spike in traffic. You suddenly have more than 75,000 users, and the system is failing. But, wait, in UAT you tested up to 100,000 concurrent users, so what’s going on?

It’s a difference in environment. Users connect to the system through sockets, socket connections require open file descriptors, and a configuration setting limits the number of file descriptors. In the UAT environment, the value found in /proc/sys/fs/file-max is 200,000, but on the production server it’s 65,535. The tests you ran in UAT didn’t test for what you’d see in production, because of the differences between the UAT and production environments.

It gets worse. After diagnosing the problem and increasing the value in the /proc/sys/fs/file-max file, all of the operations staff’s best intentions for documenting this requirement are trumped by an emergency; and later, when a new server is configured, it has the file-max value set to 65,535 again. The software is installed on that server, and the same problem will eventually once again rear its ugly head.

Remember a moment ago when I talked about needing to change property files between dev, test, staging, and production, and the impact that can have on deployments? Well, let’s say you finally have everything deployed and running, and now something changes in your infrastructure topology. Your server name, URL, or IP address changes, or you add servers for scale. If those environment configurations are in the property file, then you must re-create the deployable artifact, and you risk having additional differences creep in.

Although this might sound extreme, and I do hope that most organizations have reigned in the chaos to some degree, elements of snowflake generation persist in all but the most advanced IT departments. The bespoke environments and deployment packages clearly introduce uncertainty into the system, but accepting that deployments are going to be risky is itself a first-class problem.

**2.1.2. Risky deployments**

When are software releases scheduled at your company? Are they done during “off hours,” perhaps at 2 A.M. on Saturday morning? This practice is commonplace because of one simple fact: deployments are usually fraught with peril. It isn’t unusual for a deployment to either require downtime during an upgrade, or cause unexpected downtime. Downtime is expensive. If your customers can’t order their pizza online, they’ll likely turn to a competitor, resulting in direct revenue loss.

In response to expensive outages, organizations have implemented a host of tools and processes designed to reduce the risks associated with releasing software. At the heart of most of these efforts is the idea that we’ll do a whole bunch of up-front work to minimize the chance of failure. Months before a scheduled deployment, we begin weekly meetings to plan the “promotion into upper environments,” and change-control approvals act as the last defense to keep unforeseen things from happening in production. Perhaps the practice with the highest price tag in terms of personnel and infrastructure resources is a testing process that depends on doing trial runs on an “exact replica of production.” In principle, none of these ideas sound crazy, but in practice, these exercises ultimately place significant burdens on the deployment process itself. Let’s look at one of these practices in more detail as an example: running test deployments in an exact replica of production.

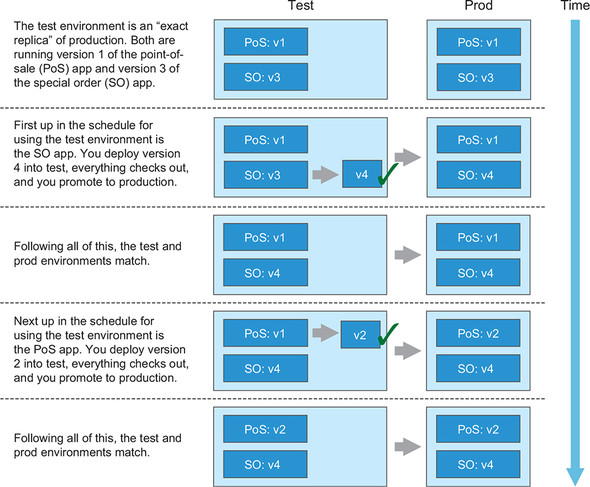
A great deal of cost is associated with establishing such a test environment. For starters, twice the amount of hardware is needed; add to that double the software, and capital costs alone grow twofold. Then there are the labor costs of keeping the test environment in alignment with production, complicated by a multitude of requirements such as the need to cleanse production data of personally identifiable information when generating testing data.

Once established, access to the test environment must be carefully orchestrated across dozens or hundreds of teams that wish to test their software prior to a production release. On the surface, it may seem like it’s a matter of scheduling, but the number of combinations of different teams and systems quickly makes it an intractable problem.

Consider a simple case in which you have two applications: a point-of-sale (PoS) system that takes payments, and a special order (SO) application that allows a customer to place an order and pay for it by using the PoS application. Each team is ready to release a new version of their application, and they must perform a test in the preproduction environment. How should these two teams’ activities be coordinated? One option is to test the applications one at a time, and although executing the tests in sequence would extend the schedule, the process is relatively tractable if all goes well with each of the tests.

[Figure 2.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig02) shows the following two steps. First, version 4 of the SO app is tested with version 1 (the old version) of the PoS app. When it’s successful, version 4 of the SO application is deployed into production. Both test and production are now running v4 of SO, and both are still running v1 of PoS. The *test* environment is a replica of *prod*. Now you can test v2 of the PoS system, and when all the tests pass, you can promote that version into production. Both application upgrades are complete, with the test and prod environments matching.

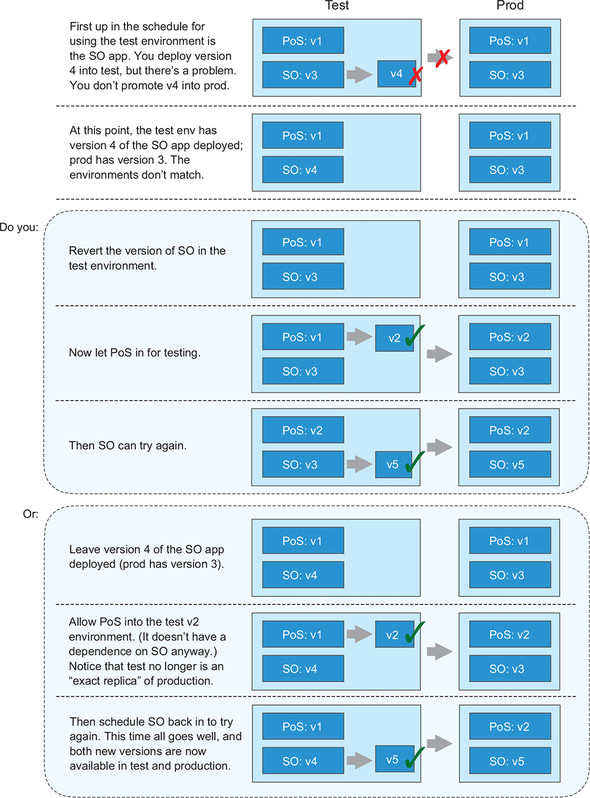
**Figure 2.2. Testing two apps in sequence is straightforward when all tests pass.**



But what happens if tests fail for the upgrade to the SO system? Clearly, you can’t deploy the new version into production. But now what do you do in the test environment? Do you revert to version 3 of SO (which takes time), even if PoS doesn’t depend on it? Was this a sequencing problem, with SO expecting PoS to already be on version 2 before it began its test? How long before SO can get back into the queue for using the test environment?

[Figure 2.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig03) shows a couple of alternatives, which get complicated quickly, even in this toy scenario. In a real setting, this becomes intractable.

**Figure 2.3. A failing test immediately complicates the process for preproduction testing.**



My goal isn’t to solve this problem here, but rather to demonstrate that even an oversimplified scenario can quickly become extraordinarily complicated. I’m sure you can imagine that when you add more applications to the mix and/or try to test new versions of multiple applications in parallel, the process becomes completely intractable. The environment that’s designed to ensure that things go well when software is deployed in production becomes a substantial bottleneck, and teams are caught between the need to get finished software out to the consumer as quickly as possible and doing it with complete confidence. In the end, it’s impossible to test exactly the scenarios that will present themselves in the production environment, and deployments remain risky business.

Risky enough, in fact, that most businesses have time periods in the year when new deployments into production aren’t permitted. For health insurance companies, it’s the open-enrollment period. In e-commerce in the United States, it’s the month between Thanksgiving and Christmas. That Thanksgiving-to-Christmas time frame is also sacred for the airline industry. The risks that persist despite efforts to minimize them make it difficult to get software deployed.

And because of this difficulty, the software running in production right now is likely to stay there for some time. We might be well aware of bugs or vulnerabilities in the apps and on the systems that are driving our customer experiences and business needs, but we must limp along until we can orchestrate the next release. For example, if an app has a known memory leak, causing intermittent crashes, we might preemptively reboot that app at regular intervals to avoid an emergency. But an increased workload against that application could cause the out-of-memory exception earlier than anticipated, and an unexpected crash causes the next emergency.

Finally, less-frequent releases lead to larger batch sizes; a deployment brings with it many changes, with equally many relationships to other parts of the system. It has been well established, and it makes intuitive sense, that a deployment that touches many other systems is more likely to cause something unexpected. Risky deployments have a direct impact on operational stability.

**2.1.3. Change is the exception**

Over the years, I’ve had dozens of conversations with CIOs and their staff who have expressed a desire to create systems that provide differentiated value to their business and their customers, but instead they’re constantly facing emergencies that draw their attention away from these innovation activities. I believe the cause of staff being in constant firefighting mode is the prevailing mindset of these long-established IT organizations: change is an exception.

Most organizations have realized the value of involving developers in initial deployments. A fair bit of uncertainty exists during fresh rollouts, and involving the team that deeply understands the implementation is essential. But at some point, responsibility for maintaining the system in production is completely handed over to the ops team, and the information for how to keep things humming is provided to them in a *runbook*. The runbook details possible failure scenarios and their resolutions, and although this sounds good in principle, on deeper reflection it demonstrates an assumption that the failure scenarios are known. But most aren’t!

The development team disengaging from ongoing operations when a newly deployed application has been stable for a predetermined period of time subtly hints at a philosophy that some point in time marks the end of change—that things will be stable from here on out. When something unexpected occurs, everyone is left scrambling. When the proverbial constant change persists, and I’ve already established that in the cloud it will, systems will persist in experiencing instability.

**2.1.4. Production instability**

All the factors I’ve covered until now inarguably hinder software from running well, but production instability itself further contributes to making deployments hard. Deployments into an already unstable environment are ill-advised; in most organizations, risky deployments remain one of the leading causes of system breakage. A reasonably stable environment is a prerequisite to new deployments.

But when the majority of time in IT is spent fighting fires, we’re left with few opportunities for deployments. Aligning those rare moments where production systems are stable with the timing of completing the complex testing cycles I talked about earlier, and the windows of opportunity shrink even further. It’s a vicious cycle.

As you can see, writing the software is only the beginning of bringing digital experiences to your customers. Curating snowflakes, allowing deployments to be risky, and treating change as an exception come together to make the job of running that software in production hard. Further insight about how these factors negatively impact operations today comes from studying well-functioning organizations—those from born-in-the-cloud companies. When you apply the practices and principles as they do, you develop a system that optimizes the entire software delivery lifecycle, from development to smooth-running operations.

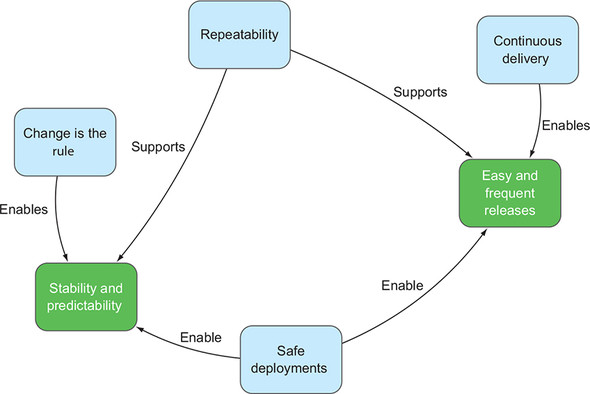
**2.2. The enablers**

A new breed of companies, those that came of age after the turn of the century, have figured out how to do things better. Google has been a great innovator, and along with some of the other internet giants, has developed new ways of running IT. With its estimated two million servers running in worldwide data centers, there’s no way that Google could’ve managed using the techniques I just described. A different way exists.

[Figure 2.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig04) presents a sketch of a system that’s almost an inverse of the bad system I described in the previous section. The goals are as follows:

* Easy and frequent releases into production
* Operational stability and predictability

**Figure 2.4. Explicit attention to these four factors develops a system of efficiency, predictability, and stability.**



You’re already familiar with the inverses of some of the factors:

* Whereas snowflakes had previously contributed to slowness and instability, repeatability supports the opposite.
* Whereas risky deployments contributed to both production instability and challenging deployments, the ability to deploy safely drives agility and stability.
* Replacing practices and software designs that depend on an unchanging environment with ones that expect constant change radically reduces time spent fighting fires.

But looking at [figure 2.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig04), you’ll notice a new entity labeled “Continuous delivery” (CD). The companies that have been most successful with the new IT operations model have redesigned their entire SDLC processes with CD as the primary driver. This has a marked effect on the ease with which deployments can happen, and the benefits ripple through the entire system.

In this section, I first explain what CD is, how basic changes in the SDLC enable CD, and the positive outcomes. I then return to the other three key enablers and describe their main attributes and benefits in detail.

**2.2.1. Continuous delivery**

Amazon may be the most extreme example of frequent releases. It’s said to release code into production for [www.amazon.com](http://www.amazon.com/) on average every second of every day. You might question the need for such frequent releases in your business, and sure, you probably don’t need to release software 86,000 times per day. But frequent releases drive business agility and enablement—both indicators of a strong organization.

Let me define *continuous delivery* by first pointing out what it isn’t. Continuous delivery doesn’t mean that every code change is deployed into production. Rather, it means that an as-new-as-possible version of the software is *deployable* at any time. The development team is constantly adding new capabilities to the implementation, but with each and every addition, they ensure that the software is ready to ship by running a full (automated!) test cycle and packaging the code for release.

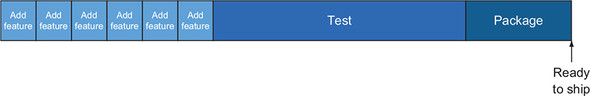
[Figure 2.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig05) depicts this cycle. Notice that there’s no “packaging” step following the “test” phase in each cycle. Instead, the machinery for packaging and deployment is built right into the development-and-test process.

**Figure 2.5. Every dev/test cycle doesn’t result in a ship; instead, every cycle results in software that’s ready to ship. Shipping then becomes a business decision.**



Contrast this with the more traditional software development practice depicted in [figure 2.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig06). A far longer single cycle is front-loaded with a large amount of software development that adds a great many features to an implementation. After a predetermined set of new capabilities has been added, an extensive test phase is completed and the software is readied for release.

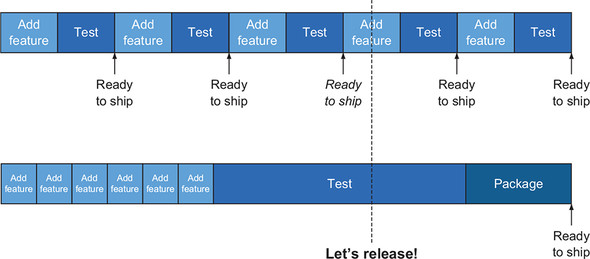
**Figure 2.6. A traditional software delivery lifecycle front-loads a lot of development work and a long testing cycle before creating the artifacts that can then be released into production.**



Let’s assume that the time spans covered by [figures 2.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig05) and [2.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig06) are the same, and that the start of each process is on the left, and the Ready to Ship point is on the far right. If you look at that rightmost point in time alone, you might not see much of a difference in outcome; roughly the same features will be delivered at roughly the same time. But if you dig under the covers, you’ll see significant differences.

First, with the former approach, the decision of when the next software release happens can be driven by the business rather than being at the mercy of a complex, unpredictable, software development process. For example, let’s say you learn that a competitor is planning a release of a product similar to yours in two weeks, and as a result, the business decides that you should make your own product immediately available. The business says, “Let’s release now!” In [figure 2.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig07), overlaying that point in time over the previous two diagrams shows a stark contrast.

**Figure 2.7. Continuous delivery is concerned with allowing business drivers, not IT readiness, to determine when software is shipped.**



Using a software development methodology that supports CD allows the Ready to Ship software of the third iteration (shown in italics) to be immediately released. True, the application doesn’t yet have all of the planned features, but the competitive advantage of being first to market with a product that has some of the features may be significant. Looking at the lower half of the figure, you see that the business is out of luck. The IT process is a blocker rather than an enabler, and the competitor’s product will hit the market first!

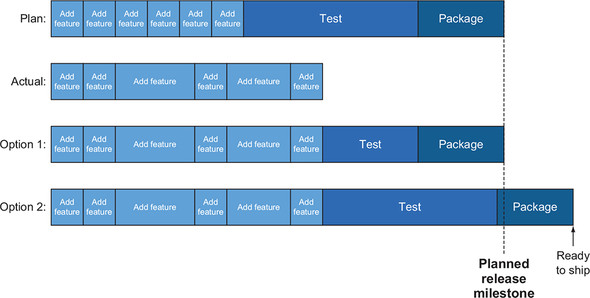
The iterative process also affords another important outcome. When the Ready to Ship versions are frequently made available to customers, it gives you an opportunity to gather feedback used to better the subsequent versions of the product. You must be deliberate about using the feedback gathered after earlier iterations to correct false assumptions or even change course entirely in subsequent iterations. I’ve seen many Scrum projects fail because they strictly adhere to plans defined at the beginning of a project, not allowing results from earlier iterations to alter those plans.

Finally, let’s admit it: we aren’t good at estimating the time it takes to build software. Part of the reason is our inherent optimism. We usually plan for the happy path, where the code works as expected immediately after the first write. (Yeah, when put like that, we see the absurdity of it right away, huh?) We also make the assumption that we’ll be fully focused on the task at hand; we’ll be cutting code all day, every day, until we get things done. And we’re probably getting pressured into agreeing to aggressive time schedules driven by market needs or other factors, usually putting us behind schedule even before we begin.

Unanticipated implementation challenges always come. Say you underestimate the effect of network latency on one part of your implementation, and instead of the simple request/response exchange that you planned for, you now need to implement a much more complex asynchronous communication protocol. And while you’re implementing this next set of features, you’re also getting pulled away from the new work to support escalations on already released versions of the software. And it’s almost never the case that your stretched goals fit within an already challenging time schedule.

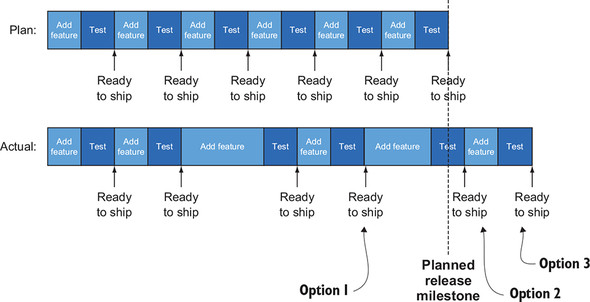
The impact these factors have on the old-school development process is that you miss your planned release milestone. [Figure 2.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig08) depicts the idealized software release plan in the first row. The second row shows the actual amount of time spent on development (longer than planned for), and the final two rows show alternatives for what you can do. One option is to stick with the planned release milestone, by compressing the testing phase, surely at the expense of software quality (the packaging phase usually can’t be shortened). A second option is to maintain the quality standards and move the release date. Neither of these options is pleasant.

**Figure 2.8. When the development schedule slips, you need to decide between two unpalatable options.**



Contrast this to the effects that “unanticipated” development delays have on a process that implements many shorter iterations. As depicted in [figure 2.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig09), you again see that your planned release milestone is expected to come after six iterations. When the actual implementation takes longer than expected, you see that you’re presented with some new options. You can either release on schedule with a more limited set of features (option 1), or you can choose a slight or longer delay for the next release (options 2 and 3). The key is that the business is presented with a far more flexible and palatable set of options. And when, through the system I’m presenting in this section, you make deployments less risky and therefore deploy more frequently, you can complete those two releases in rapid succession.

**Figure 2.9. Shorter iterations designed for continuous delivery allow for an agile release process while maintaining software quality.**



To net it all out, lengthy release cycles introduce a great deal of risk into the process of bringing digital products to consumers. The business lacks the ability to control when products are released to the market, and the organization as a whole is often in the awkward position of trading off near-term market pressures with long-term goals of software quality and ability to evolve.

|  |
| --- |
|  |

**Note**

Short iterations release a great deal of tension from the system. Continuous delivery allows business drivers to determine how and when products are brought to market.

|  |
| --- |
|  |

I’ve talked about continuous delivery first, and at relative length, because it truly is at the core of a new, functional system of software development and operations. If your organization isn’t yet embracing practices such as these, this is where your initial efforts should be placed. Your ability to change the way that you bring software to market is hindered without such changes. And even the structure of the software you build, which is what this book is about, is linked to these practices in both subtle and direct ways. Software architecture is what this book is about, and we’ll cover that in depth throughout.

Now let’s go back to [figure 2.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig04) and study the other factors that support our operational goals of easy, frequent releases and software stability.

**2.2.2. Repeatability**

In the previous section, I talked about the detrimental effect of variability, or as we often call them, *snowflakes*, on the workings of IT. They make things hard to deploy because you must constantly adjust to differences in both the environments into which you’re deploying, and in the variability of the artifacts you’re deploying. That same inconsistency makes it extremely difficult to keep things running well once in production, because every environment and piece of software gets special treatment anytime something changes. Drift from a known configuration is a constant threat to stability when you can’t reliably re-create the configuration that was working before a crash.

When you turn that negative into a positive in your enabling system, the key concept is repeatability. It’s analogous to the steps in an assembly line: each time you attach a steering wheel to a car, you repeat the same process. If the conditions are the same within some parameters (I’ll elaborate on this more in a moment), and the same process is executed, the outcome is predictable.

The benefits of repeatability on our two goals—getting things deployed and maintaining stability—are great. As you saw in the previous section, iterative cycles are essential to frequent releases, and by removing the variability from the dev/test process that happens with each turn of the crank, the time to deliver a new capability within the iteration is compressed. And once running in production, whether you’re responding to a failure or increasing capacity to handle greater volumes, the ability to stamp out deployments with complete predictability relieves tremendous stress from the system.

*How do we then achieve this sought-after repeatability?* One of the advantages of software is that it’s easy to change, and that malleability can be done quickly. But this is also exactly what has invited us to create snowflakes in the past. To achieve the needed repeatability, you must be disciplined. In particular, you need to do the following:

* Control the environments into which you’ll deploy the software
* Control the software that you’re deploying—also known as the *deployable artifact*
* Control the deployment processes

**Control the environment**

In an assembly line, you control the environment by laying out the parts being assembled and the tools used for assembly in exactly the same way—no need to search for the three-quarter-inch socket wrench each time you need it, because it’s always in the same place. In software, you use two primary mechanisms to consistently lay out the context in which the implementation runs.

First, you must begin with standardized machine images. In building up environments, you must consistently begin with a known starting point. Second, changes applied to that base image to establish the context into which your software is deployed *must be coded*. For example, if you begin with a base Ubuntu image and your software requires the Java Development Kit (JDK), you’ll script the installation of the JDK into the base image. The term often used for this latter concept is *infrastructure as code*. When you need a new instance of an environment, you begin with the base image and apply the script, and you’re guaranteed to have the same environment each time.

Once established, any changes to an environment must also be equally controlled. If operations staff routinely ssh into machines and make configuration changes, the rigor you’ve applied to setting up the systems is for naught. Numerous techniques can be used to ensure control after initial deployment. You may not allow SSH access into running environments, or if you do, automatically take a machine offline as soon as someone has ssh’d in. The latter is a useful pattern in that it allows someone to go into a box to investigate a problem, but doesn’t allow for any potential changes to color the running environment. If a change needs to be made to running environments, the only way for this to happen is by updating the standard machine image as well as the code that applies the runtime environment to it—both of which are controlled in a source code control system or something equivalent.

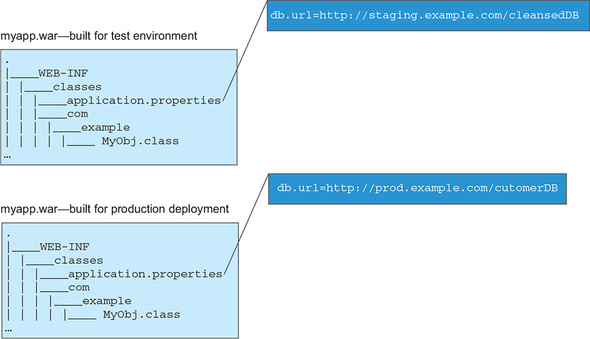
Who is responsible for the creation of the standardized machine images and the infrastructure-as-code varies, but as an application developer, it’s essential that you use such a system. Practices that you apply (or don’t) early in the software development lifecycle have a marked effect on the organization’s ability to efficiently deploy and manage that software in production.

**Control the deployable artifact**

Let’s take a moment to acknowledge the obvious: there are always differences in environments. In production, your software connects to your live customer database, found at a URL such as <http://prod.example.com/cutomerDB>; in staging, it connects to a copy of that database that has been cleansed of personally identifiable information and is found at <http://staging.example.com/cleansedDB>; and during initial development, there may be a mock database that’s accessed at http://localhost/mockDB. Obviously, credentials differ from one environment to the next. How do you account for such differences in the code you’re creating?

I know you aren’t hardcoding such strings directly into your code (right?). Likely, you’re parameterizing your code and putting these values into some type of a property file. This is a good first step, but often a problem remains: the property files, and hence the parameter values for the different environments, are often compiled into the deployable artifact. For example, in a Java setting, the application.properties file is often included in the JAR or WAR file, which is then deployed into one of the environments. And therein lies the problem. When the environment-specific settings are compiled in, the JAR file that you deploy in the test environment is different from the JAR file that you deploy into production; see [figure 2.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig10).

**Figure 2.10. Even when environment-specific values are organized into property files, including property files in the deployable artifact, you’ll have different artifacts throughout the SDLC.**



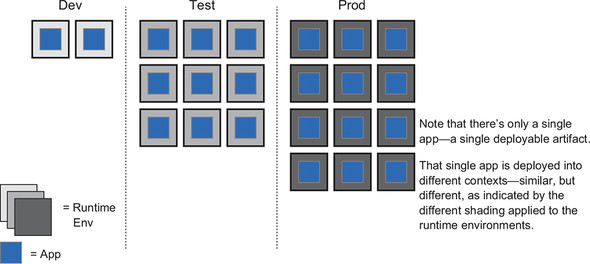
As soon as you build different artifacts for different stages in the SDLC, repeatability may be compromised. The discipline for controlling the variability of that software artifact, ensuring that the only difference in the artifacts is the contents of the property files, must now be implanted into the build process itself. Unfortunately, because the JAR files are different, you can no longer compare file hashes to verify that the artifact that you’ve deployed into the staging environment is exactly the same as that which you’ve deployed into production. And if something changes in one of the environments, and one of the property values changes, you must update the property file, which means a new deployable artifact and a new deployment.

For efficient, safe, and repeatable production operations, it’s essential that a single deployable artifact is used through the entire SDLC. The JAR you build and run through regression tests during development is the *exact* JAR file deployed into the test, staging, and production environments. To make this happen, the code needs to be structured in the right way. For example, property files don’t carry environment-specific values, but instead define a set of parameters for which values may later be injected. You can then bind values to these parameters at the appropriate time, drawing values from the right sources. It’s up to you as the developer to create implementations that properly abstract the environmental variability. Doing this allows you to create a single deployable artifact that can be carried through the entire SDLC, bringing with it agility and reliability.

**Control the process**

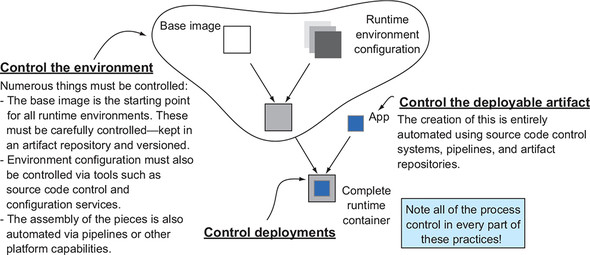
Having established environment consistency, and the discipline of creating a single deployable artifact to carry through the entire software development lifecycle, what’s left is ensuring that these pieces come together in a controlled, repeatable manner. [Figure 2.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig11) depicts the desired outcome: in all stages of the SDLC, you can reliably stamp out exact copies of as many running units as needed.

**Figure 2.11. The desired outcome is to be able to consistently establish apps running in standardized environments. Note that the app is the same across all environments; the runtime environment is standardized within an SDLC stage.**



This figure has no snowflakes. The deployable artifact, the app, is exactly the same across all deployments and environments. The runtime environment has variation across the different stages, but (as indicated by the different shades of the same gray coloring) the base is the same and has only different configurations applied, such as database bindings. Within a lifecycle stage, all the configurations are the same; they have exactly the same shade of gray. Those antisnowflake boxes are assembled from the two controlled entities I’ve been talking about: standardized runtime environments and single deployable artifacts, as seen in [figure 2.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig12).

**Figure 2.12. The assembly of standardized base images, controlled environment configurations, and single deployable artifacts is automated.**



A whole lot is under the surface of this simple picture. What makes a good base image, and how is it made available to developers and operators? What is the source of the environment configuration, and when is it brought into the application context? Exactly when is the app “installed” into the runtime context? I’ll answer these questions and many more throughout the book, but at this juncture my main point is this: the only way to draw the pieces together in a manner that ensures consistency is to automate.

Although the use of continuous integration tools and practices is fairly ubiquitous in the development phase of writing software (for example, a build pipeline compiles checked-in code and runs some tests), its use in driving the entire SDLC isn’t as widely adopted. But the automation must carry all the way from code check-in, through deployments, into test and production environments.

And when I say it’s all automated, I mean *everything*. Even when you aren’t responsible for the creation of the various bits and pieces, the assembly must be controlled in this manner. For example, users of Pivotal Cloud Foundry, a popular cloud-native platform, use an API to download new “stem cells,”**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fn2)**]** the base images into which apps are deployed, from a software distribution site, and use pipelines to complete the assembly of the runtime environment and the application artifact. Another pipeline does the final deployment into production. In fact, when deployments into production also happen via pipelines, servers aren’t touched directly by humans, something that’ll make your chief security officer (and other control-related personnel) happy.

***2***

*See Pivotal’s API Documentation page at*[*https://network.pivotal.io/docs/api*](https://network.pivotal.io/docs/api)*for more information.*

But if you’ve totally automated things all the way to deployment, how do you ensure that these deployments are safe? This is another area that requires a new philosophy.

**2.2.3. Safe deployments**

Earlier I talked about risky deployments and that the most common mechanism that organizations use as an attempt to control the risk is to put in place expansive and expensive testing environments with complex and slow processes to govern their use. Initially, you might think that there’s no alternative, because the only way to know that something works when deployed into production is to test it first. But I suggest that it’s more a symptom of what Grace Hopper said was the most dangerous phrase: “We’ve always done it this way.”

The born-in-the-cloud-era software companies have shown us a new way: they experiment in production. Egad! What am I talking about?! Let me add one word: they *safely* experiment in production.

Let’s first look at what I mean by *safe experimentation* and then look at the impact it has on our goals of easy deployments and production stability.

When trapeze artists let go of one ring, spin through the air, and grasp another, they most often achieve their goal and entertain spectators. No question about it, their success depends on the right training and tooling, and a whole load of practice. But acrobats aren’t fools; they know that things sometimes go wrong, so they perform over a safety net.

When you experiment in production, you do it with the right safety nets in place. Both operational practices and software design patterns come together to weave that net. Add in solid software-engineering practices such as test-driven development, and you can minimize the chance of failure. But eliminating it entirely isn’t the goal. Expecting failure (and failure will happen) greatly lessens the chances of it being catastrophic. Perhaps a small handful of users will receive an error message and need to refresh, but overall the system remains up and running.

|  |
| --- |
|  |

**Tip**

Here’s the key: everything about the software design and the operational practices allows you to easily and quickly pull back the experiment and return to a known working state (or advance to the next one) when necessary.

|  |
| --- |
|  |

This is the fundamental difference between the old and the new mindset. In the former, you tested extensively before going to production, believing you’d worked out all the kinks. When that delusion proved incorrect, you were left scrambling. With the new, you plan for failure, intentionally creating a retreat path to make failures a non-event. This is empowering! And the impact on your goals, easier and faster deployments, and stability after you’re up and running is obvious and immediate.

First, if you eliminate the complex and time-consuming testing process that I described in [section 2.1.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02lev2sec2), and instead go straight to production following basic integration testing, a great deal of time is cut from the cycle and, clearly, releases can occur more frequently. The release process is intentionally designed to encourage its use and involves little ceremony to begin. And having the right safety nets in place allows you to not only avert disaster, but to quickly return to a fully functional system in a matter of seconds.

When deployments come without ceremony and with greater frequency, you’re better able to address the failings of what you’re currently running in production, allowing you to maintain a more stable system as a whole.

Let’s talk a bit more about what that safety net looks like, and in particular, the role that the developer, architect, and application operators play in constructing it. You’ll look at three inextricably linked patterns:

* Parallel deployments and versioned services
* Generation of necessary telemetry
* Flexible routing

In the past, a deployment of version *n* of some software was almost always a replacement of version *n* – 1. In addition, the things we deployed were large pieces of software encompassing a wide range of capabilities, so when the unexpected happened, the results could be catastrophic. An entire mission-critical application could experience significant downtime, for example.

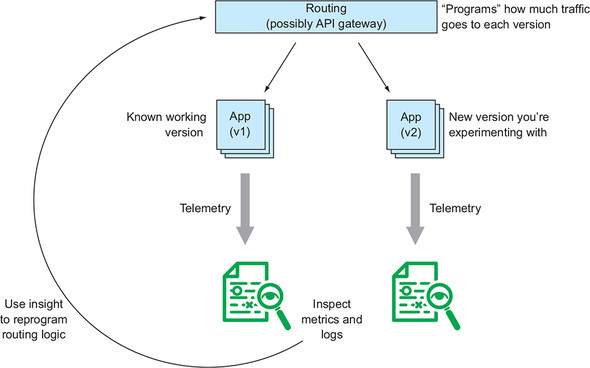
At the core of your safe deployment practices is parallel deployment. Instead of completely replacing one version of running software with a new version, you keep the known working version running as you add a new version to run alongside it. You start out with only a small portion of traffic routed to the new implementation, and you watch what happens. You can control which traffic is routed to the new implementation based on a variety of available criteria, such as where the requests are coming from (either geographically or what the referring page is, for example) or who the user is.

To assess whether the experiment is yielding positive results, you look at data. Is the implementation running without crashing? Has new latency been introduced? Have click-through rates increased or decreased?

If things are going well, you can continue to increase the load directed at the new implementation. If at any time things aren’t happy, you can shift all the traffic back to the previous version. This is the retreat path that allows you to experiment in production.

[Figure 2.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig13) shows how the core practice works.

**Figure 2.13. Data tells you how parallel deployments of multiple versions of your apps are operating. You use that data to program control flows to those apps, supporting safe rollouts of new software in production.**



None of this can be done if proper software engineering disciplines are ignored, or applications don’t embody the right architectural patterns. Some of the keys to enable this form of A/B testing are as follows:

* Software artifacts must be versioned, and the versions must be visible to the routing mechanism to allow it to appropriately direct traffic. Further, because you’ll be analyzing data to determine whether the new deployment is stable and achieving the desired outcomes, all data must be associated with the appropriate version of the software in order to make the proper comparisons.
* The data used to analyze how the new version is functioning takes a variety of forms. Some metrics are completely independent of any details of the implementation; for example, the latency between a request and response. Other metrics begin to peer into the running processes, reporting on things such as the number of threads or memory being consumed. And finally, domain-specific metrics, such as the average total purchase amount of an online transaction, may also be used to drive deployment decisions. Although some of the data may automatically be provided by the environment in which the implementation is running, you won’t have to write code to produce it. The availability of data metrics is a first-class concern. I want you to think about producing data that supports experimentation in production.
* Clearly, routing is a key enabler of parallel deployments, and the routing algorithms are pieces of software. Sometimes the algorithm is simple, such as sending a percentage of all the traffic to the new version, and the routing software “implementation” can be realized by configuring some of the components of your infrastructure. Other times you may want more-sophisticated routing logic and need to write code to realize it. For example, you may want to test some geographically localized optimizations and want to send requests only from within the same geography to the new version. Or perhaps you wish to expose a new feature only to your premium customers. Whether the responsibility for implementing the routing logic falls to the developer or is achieved via configuration of the execution environment, routing is a first-class concern for the developer.
* Finally, something I’ve already hinted at is creating smaller units of deployment. Rather than a deployment encompassing a huge portion of your e-commerce system—for example, the catalog, search engine, image service, recommendation engine, shopping cart, and payment-processing module all in one—deployments should have a far smaller scope. You can easily imagine that a new release of the image service poses far less risk to the business than something that involves payment processing. Proper componentization of your applications—or as many would call it today, a microservices-based architecture—is directly linked to the operability of digital solutions.**[**[**3**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fn3)**]**

***3***

*Google provides more details in its “Accelerate: State of DevOps” report, available at*[*http://mng.bz/vNap*](http://mng.bz/vNap)*.*

Although the platform your applications run on provide some of the necessary support for safe deployments (and I’ll talk more about this in [chapter 3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03)), all four of these factors—versioning, metrics, routing, and componentization—are things that you, as a developer, must consider when you design and build your cloud-native application. There’s more to cloud-native software than these things (for example, designing bulkheads into your architecture to keep failures from cascading through the entire system), but these are some of the key enablers of safe deployments.

**2.2.4. Change is the rule**

Over the last several decades, we’ve seen ample evidence that an operational model predicated on a belief that our environment changes only when we intentionally and knowingly initiate such changes doesn’t work. Reacting to unexpected changes dominates the time spent by IT, and even traditional SDLC processes that depend on estimates and predictions have proven problematic.

As we’re doing with the new SDLC processes I’ve been describing throughout this chapter, building muscle that allows you to adapt when change is thrust upon you affords far greater resilience. What’s subtle is identifying what those muscles are when it comes to stability and predictability for production systems. This concept is a bit tricky, a bit “meta” if you will; please bear with me a moment.

The trick isn’t to get better at predicting the unexpected or allocating more time for troubleshooting. For example, allocating half of a development team’s time to responding to incidents does nothing to address the underlying cause of the firefighting. You respond to an outage, get everything in working order, and you’re done—until the next incident.

“*Done*.”

This is the root of the problem. You believe that after you’re finished with a deployment, responding to an incident, or making a firewall change, you’ve somehow completed your work. The idea that you’re “done” inherently treats change as something that causes you to become not done.

|  |
| --- |
|  |

**Tip**

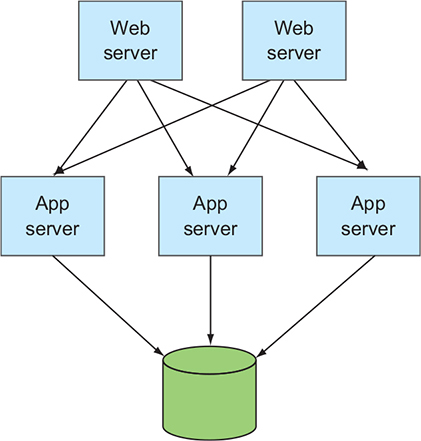
You need to let go of the notion of ever being done.

|  |
| --- |
|  |

Let’s talk about *eventual consistency*. Rather than creating a set of instructions that brings a system into a “done” state, an eventually consistent system never expects to be done. Instead, the system is perpetually working to achieve equilibrium. The key abstractions of such a system are the desired state and the actual state.

The *desired state* of a system is what you want it to look like. For example, say you want a single server running a relational database, three application servers running RESTful web services, and two web servers delivering rich web applications to the users. These six servers are properly networked, and firewall rules are appropriately set. This topology, as shown in [figure 2.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig14), is an expression of the desired state of the system.

**Figure 2.14. The desired state of your deployed software**

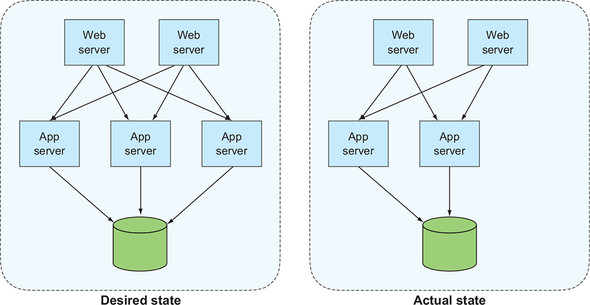


You’d hope that, at some point, even most of the time, you have that system entirely established and running well, but you’ll never assume that things remain as you left them immediately following a deployment. Instead, you treat the *actual state*, a model of what’s currently running in your system, as a first-class entity, constructing and maintaining it by using some of the metrics you already considered in this chapter.

The eventually consistent system then constantly compares the *actual state* to the *desired state*, and when there’s a deviation, performs actions to bring them back into alignment. For instance, let’s say that you lose an application server from the topology laid out in [figure 2.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig14). This could happen for any number of reasons—a hardware failure, an out-of-memory exception coming from the app itself, or a network partition that cuts off the app server from other parts of the system.

[Figure 2.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig15) depicts both the desired state and the actual state. The actual state and desired state clearly don’t match. To bring them back into alignment, another application server must be spun up and networked into the topology, and the application must be installed and started thereon (recall earlier discussions around repeatable deployments).

**Figure 2.15. When the actual state doesn’t match the desired state, the eventually consistent system initiates actions to bring them back into alignment.**



For those of you who previously might not have done much with eventual consistency, this might feel a bit like rocket science. An expert colleague avoids using the term *eventual consistency* because he worries that it’ll invoke fear in our customers. But systems built on this model are increasingly common, and many tools and educational materials can assist in bringing such solutions to fruition.

And I’ll tell you this: it’s absolutely, totally, completely essential to running applications on the cloud. I’ve said it before: things are always changing, so better to embrace that change than to react to it. You shouldn’t fear eventual consistency. You should embrace it.

Let me clarify something. Although the system I’m referring to here isn’t necessarily entirely automated, having a platform that implements the core portions of the paradigm is required (I’ll say more about the role of the platform in the next chapter). What I want you to do is design and build your software in a manner that allows a self-healing system to adapt to the constant change inflicted upon it. Teaching you how to do this is the aim of this book.

Software designed to remain functional in the face of constant change is the Holy Grail, and the impact on system stability and reliability is obvious. A self-healing system maintains higher uptime than one that requires human intervention each time something goes wrong. And treating a deployment as an expression of a new desired state greatly simplifies it and reduces risk. Adopting a mindset that change is the rule fundamentally alters the nature of managing software in production.

**Summary**

* In order for value to be realized from the code you write, you need to be able to do two things: get it deployed easily and frequently, and keep it running well in production.
* Missing the mark on either of these tasks shouldn’t be blamed on developers or operators. Instead, the “blame” rests with a failing system.
* The system fails because it allows bespoke solutions, which are hard to maintain; creates an environment that makes the act of deploying software inherently risky; and treats changes in the software and environment as an exception.
* When deployments are risky, they’re performed less frequently, which only serves to make them even riskier.
* You can invert each of these negatives—focusing on repeatability, making deployments safe, and embracing change—and create a system that supports rather than hinders the practices you desire.
* Repeatability is at the core of optimized IT operations, and automation applies not only to the software build process, but also to the creation of runtime environments and the deployment of applications.
* Software design patterns as well as operational practices expect the constant change in cloud-based environments.
* The new system depends on a highly iterative SDLC that supports continuous delivery practices.
* Continuous delivery is what a responsive business needs to compete in today’s markets.
* Finer granularity throughout the system is key. Shorter development cycles and smaller application components (microservices) account for significant gains in agility and resilience.
* Eventual consistency reigns supreme in a system where change is the rule.