Author Picks

FREE

Principles of Cloud Design

Chapters selected by David Clinton

manning

Learn Amazon Web Services in a Month of Lunches

by David Clinton

ISBN 9781617294440
328 pages
$39.99
August 2017

Production Ready Serverless

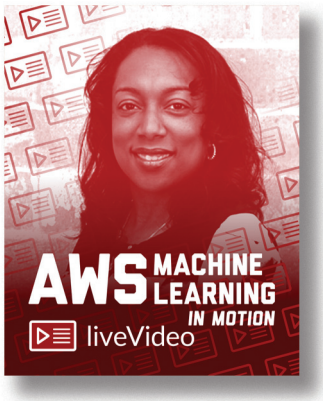
by Yan Cui

Course duration: 10h
$79.99
December 2018

Amazon Web Services in Action, Second Edition

by Michael Wittig and Andreas Wittig

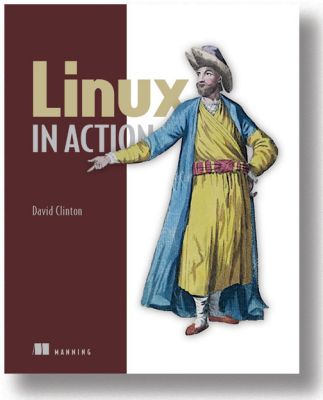
ISBN 9781617295119
528 pages
$54.99
September 2018



AWS Machine Learning in Motion

by Kesha Williams

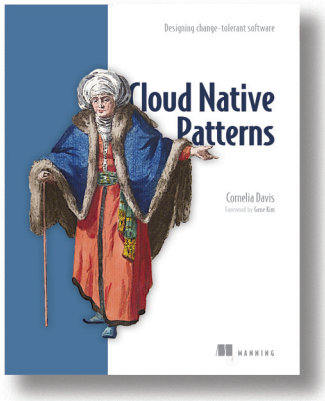
Course duration: 3h 42m
$39.99
October 2018



Linux in Action

by David Clinton

ISBN 9781617294938
384 pages
$39.99
August 2018



Cloud Native Patterns

by Cornelia Davis

ISBN 9781617294297
400 pages
$39.99
May 2019

Principles of Cloud Design

Chapters Selected by David Clinton

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617296628
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 24 23 22 21 20 19

about the autor

David Clinton is a System Administrator and Linux Server Professional with years of experience teaching IT subjects. He's the author of “Learn Amazon Web Services in a Month of Lunches”, “Linux in Action”, and dozens of video courses on Linux administration, Amazon Web Services, and server security. He can be found at https://bootstrap-it.com.

contents

introduction

In the mad rush to move our IT infrastructure over to cloud platforms, it's important to remember that "the cloud" is more than a bunch of servers sitting in someone else's building rather than your own. This brand-new world has some brand-new rules. Although some of the things you'll do on a public or private cloud work much the same as they do locally, unless you learn to design with a cloud mentality, you'll likely miss many of the biggest benefits... particularly as your deployments grow to enterprise scale.

The three chapters that make up this e-book highlight three ways that smart design feeds successful cloud deployments. The principles you'll encounter here can help you squeeze greater efficiency and effectiveness from your cloud infrastructure.

Static or elastic? Coupled or decoupled dependencies? Public cloud, private cloud, or hybrid? If you're planning a cloud deployment of any kind, these questions stare you in the face. Read on and learn how to answer them.

"Keeping Ahead of user Demand" from the Manning book "Learn Amazon Web Services in a Month of Lunches" introduces you to the theory and practice behind high availability, scalability, and elasticity. Only by understanding the function played by each of those concepts—including how scalability and elasticity differ—can you fully benefit from the powerful software driving the major cloud engines.

Chapter 13 from *Learn Amazon Web Services in a Month of Lunches* by David Clinton

Keeping ahead of user demand

This is a bit of a transition chapter. Chapter 12 completed our survey of AWS's core deployment services. That means you're now familiar with the following:

- EC2 instances, Amazon Machine Images (AMIs), and the peripheral tools that support their deployment, such as security groups and EBS volumes
- Incorporating databases into applications, both on-instance and through the managed RDS service
- Using S3 buckets to deliver media files through your EC2 applications and for server backup storage
- Controlling access to your AWS resources with Identity and Access Management (IAM)
- Managing growing resource sets by intelligently applying tags
- Accessing resources using either the browser interface or the AWS command-line interface (CLI)

All of these things are represented in the schematic shown in figure 13.1. That alone would easily justify the time and energy you've invested in this book. Now we're going to shift our focus and explore some best practices for application optimization.

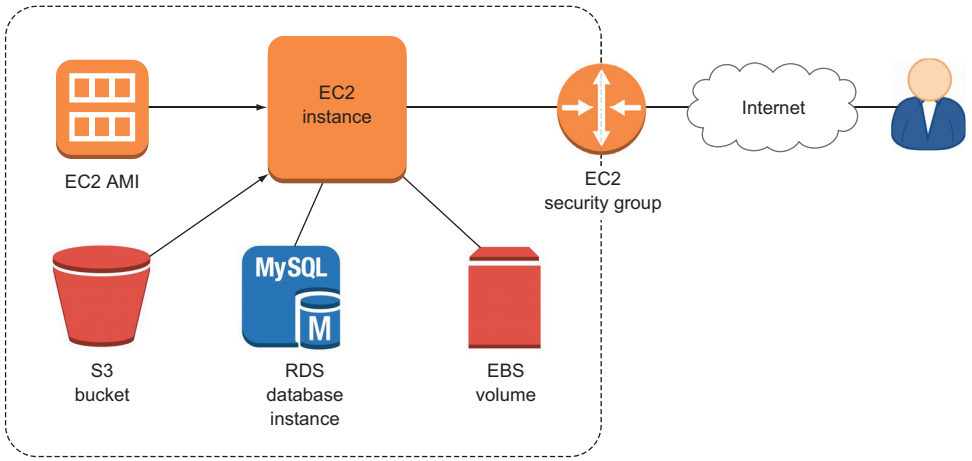


Figure 13.1 **This is the kind of application infrastructure you should be able to build on your own, having read the first dozen chapters of this book.**

But things are running nicely—who needs optimization? Well, as customer demand on your WordPress site continues to grow, you'll care, and in a big way. You see, for some reason—perhaps related to the fact that you discount the price of your product by 75% for half an hour each evening—most customers arrive in the early evening, local time. The single server you've been running is largely unused most of the day, but it melts under the pressure of thousands of visits squeezed into such a short stretch of time.

And then there's a question one of the guys in the office asked the other day: "Our entire business is running on a single web server. What happens if it goes down?" What indeed.

You could provision four or five extra servers and run them full time. That way, you'd be covered for the high-volume periods and for the failure of any one server. But that approach would involve colossal waste, because for much of each day you'd be paying for most of the instances to sit idle. Nor would it necessarily be much help in the event of a *network* failure, which would likely cut connectivity to *all* the servers at the same time.

You could address the customer demand issue by arranging for someone to be at the office every evening to manually fire up as many extra servers as needed; but you asked around, and no one volunteered. And besides, the best way to ensure that a daily job won't get done is to assume that an admin will remember to do it.

13.1 Automating high availability

Alternatively, you could spend some time incorporating high availability capability into your setup and let software quietly and efficiently manage things. This will be the subject of the next few chapters. You'll learn to use AWS's geographically remote availability zones to make total application failure much less likely; load balancing to coordinate between parallel servers and monitor their health; and auto scaling to let AWS automatically respond to the peaks and valleys of changing demand by launching and shutting down instances according to need.

> **NOTE** *High availability* is any server resource configuration that allows a system to remain functioning and accessible for as close to 100% of the time as possible. The basic goal can be achieved through combinations that can include redundancy, replication, failover protocols, monitoring, and load balancing.

Figure 13.2 will help you visualize how all that infrastructure can be made highly available through the magic of network segmenting, auto scaling, and load balancing. Although you probably aren't familiar with many of the tools and relationships represented in the diagram, you should make a mental note of at least a few key points:

1. A virtual private cloud (VPC) encompasses all the AWS resources in your application deployment.
2. There are two kinds of subnets—private and public—that can be located in separate availability zones and are used to manage and, where needed, isolate resources.
3. Security-group rules control the movement of data between resources.
4. The EC2 AMI acts as a template for replicating precise OS environments.
5. The S3 bucket can store and deliver data, both for backup and for delivery to users.
6. The EBS volumes act as data volumes (like hard drives) for an instance.
7. The auto scaler permits automatic provisioning of more (or fewer) instances to meet changing demands on an application.
8. The load balancer routes traffic among multiple servers to ensure the smoothest and most efficient user experience.

As you've probably noticed, the *E* in many AWS service names (EC2, ECS, EFS, EMR, and so on) doesn't stand for *electronic* the way it does in the names of some older technologies, like *email*; rather, it stands for *elastic*. You can be excused for wondering just what about the AWS vision of cloud computing is so elastic.

But before I answer that question, it may be useful to talk about cloud computing in general. Understanding what makes the cloud unique is essential for taking full advantage of all that it has to offer.

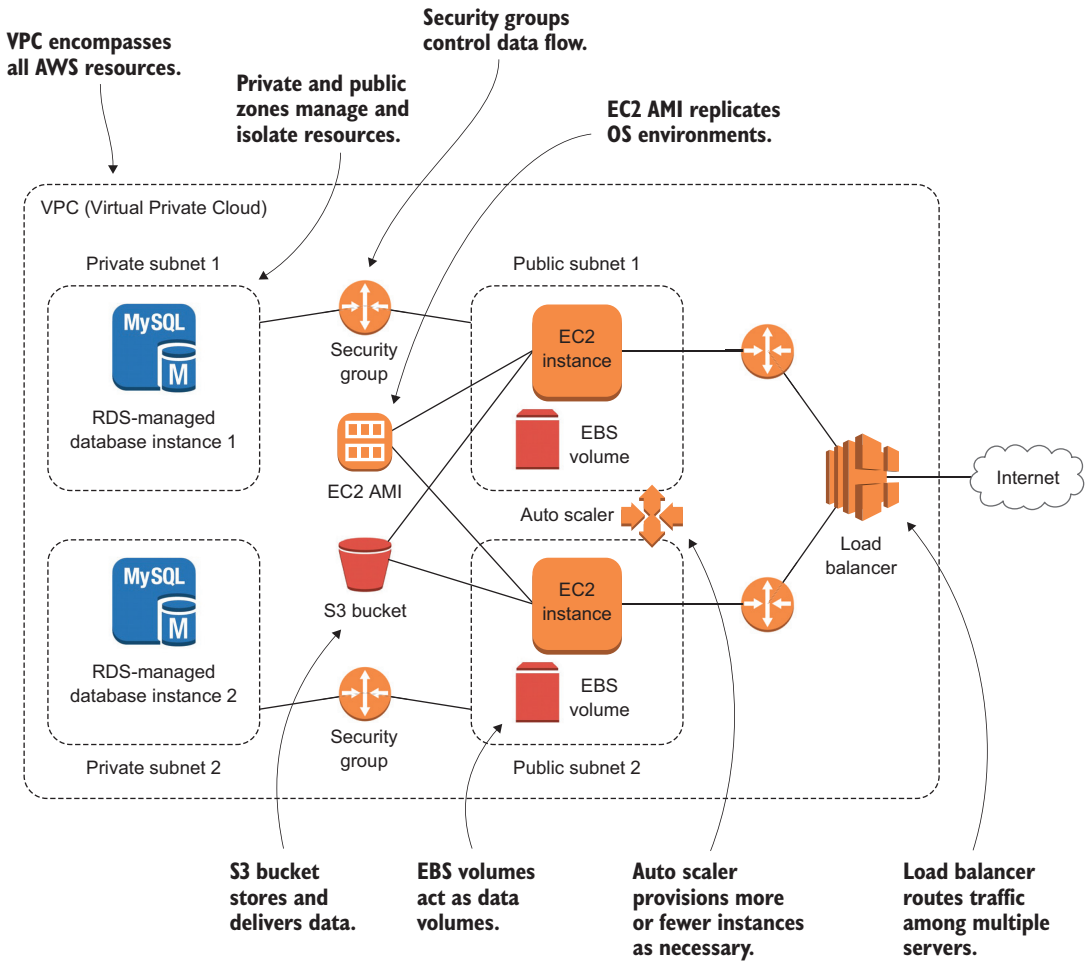


Figure 13.2 An illustration of how AWS data and security services work together to allow an EC2 instance to deliver its application

13.2 Cloud computing

The U.S. National Institute of Standards and Technology (NIST) defines *cloud computing* as services that offer users all of these five qualities:

- *On-demand self-service*—Customers can access public cloud resources whenever needed and without having to order them through a human representative.
- *Broad network access*—Cloud resources are accessible from any network-connected (that is, internet) location.
- *Resource pooling*—Cloud providers offer a multitenant model, whereby individual customers can safely share resources with each other; and dynamic resource assignment, through which resources can be allocated and deallocated according to customer demand.

- *Rapid elasticity*—Resource availability and performance can be automatically increased or decreased to meet changing customer demand.
- *Measured service*—Customers can consume services at varying levels through a single billing period and are charged only for those resources they actually use.

These five qualities describe a deeply flexible, highly automated system whose elements can be freely mixed and matched to provide the efficient, cost-effective service. But a great deal of what makes this possible is the existence of integrated systems that can dynamically adjust themselves based on what's going on around them. These adjustments are examples of elastic behavior.

13.3 Elasticity vs. scalability

Elasticity is a system's ability to monitor user demand and automatically increase and decrease deployed resources accordingly. *Scalability*, by contrast, is a system's ability to monitor user demand and automatically increase and decrease … wait, didn't I just say that about elasticity?

It's complicated. The two terms are sometimes used interchangeably, but I think it's worthwhile distinguishing between them. Bear in mind that the way I explain the relationship between these two ideas is by no means the last word on the subject—look around, and you'll find some other approaches. But in the context of understanding how AWS works, my spin should be useful.

What makes an elastic band *elastic* is partly its ability to stretch under pressure, but also the way it quickly returns to its original size when the pressure is released. In AWS terms, that would mean the way, for instance, EC2 makes instances available to you when needed but lets you drop them when they're not, and charges you only for uptime (see figure 13.3).

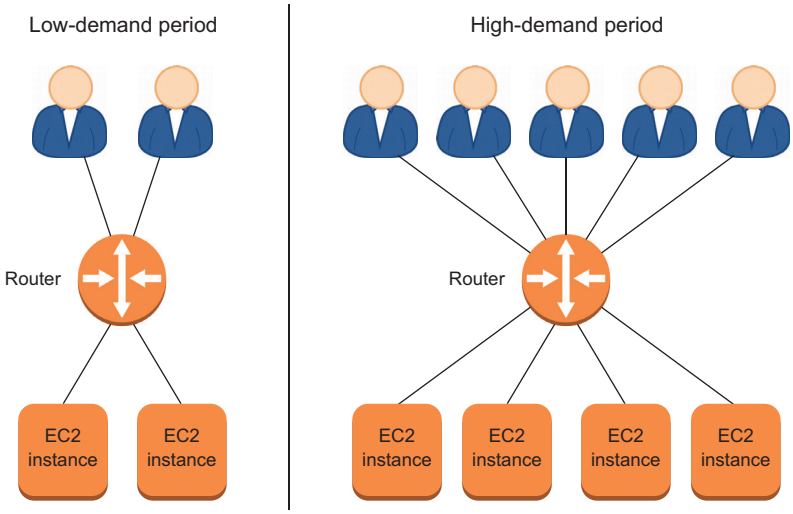


Figure 13.3 Elasticity allows for systems to dynamically add or remove resources to meet changing demand.

Scalability describes the way a system is *designed* to meet changing demand. That might include the fact that you have 24-hour access to any resources you might need (which, of course, is an elastic feature), but it also means the *underlying design* supports rapid, unpredictable changes. As an example, software that's scalable can be easily picked up and dropped onto a new server—possibly in a new network environment—and run without any manual configuration. Similarly, as shown in figure 13.4, the composition of a scalable infrastructure can be quickly changed in a way that all the old bits and pieces immediately know how to work together with the new ones.

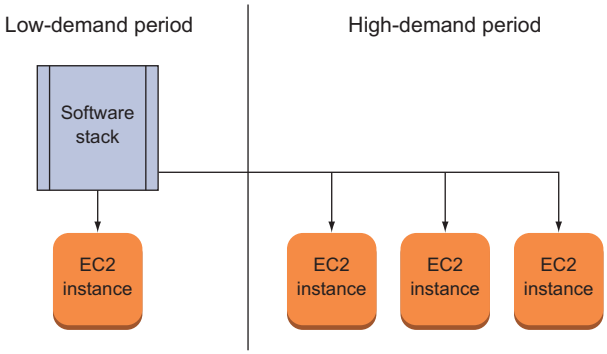


Figure 13.4 Scalable software can be easily copied for use in multiple servers deployed in multiple network environments.

With that in mind, we can say that Amazon's EC2 is not only elastic but, because its elements—instances, storage volumes, security groups, and so on—can be smoothly dropped into and out of running infrastructures, also very scalable. Ah, but what *kind* of scalable? There are two:

- *Horizontal scaling* is *scaling out*: you add more lightweight server nodes (or *instances*) to meet growing demand.
- *Vertical scaling* is *scaling up*: you move your application from a single lightweight server to one with greater compute capacity.

It's certainly possible to transfer AWS-based applications from lighter to heavier servers, and for some payloads—like many high-load transaction databases—it's preferred. But in an AWS context, if you hear a conjugation of the word *scale*, the odds are that it's referring to horizontal scaling.

If you want a more reliable, responsive, public-facing application (and who doesn't?), you should definitely stay tuned as we work through the book's second part, which focuses on optimizing your existing infrastructure.

Definitions

- *Cloud computing*—Networked services offering self-service, resource pooling, elasticity, and metered billing
- *Elasticity*—The ability to increase and decrease available compute resources to meet changing demands
- *Scalability*—The ability of software or infrastructure to adapt to changes in service volume
- *Scaling out*—The addition of new server nodes to handle increased demand (horizontal scaling)
- *Scaling up*—The adoption of a more powerful server node to handle increased demand (vertical scaling)

"Decoupling Your Infrastructure: ELB and SQS" from "Amazon Web Services in Action" teaches you how to tame the multi-headed monster of the software-driven cloud. If, through the magic of scaling, your resources are being launched and killed off without warning, how can they find each other? The answer: decouple. Rather than connecting resources by hard coding absolute dependencies, abstract the relationships and let smart services like Amazon's Elastic Load Balancer and Simple Queue Service keep track.

Chapter 12 from *Amazon Web Services in Action*
by Michael Wittig and Andreas Wittig

Decoupling your infrastructure: ELB and SQS

This chapter covers

- The reasons for decoupling a system
- Synchronous decoupling with load balancers
- Asynchronous decoupling with message queues

Imagine that you want some advice on using AWS from us, and therefore we plan to meet in a café. To make this meeting successful, we must

- Be available at the same time
- Be at the same place
- Find each other at the café

The problem with our meeting is that it's tightly coupled to a location. We can solve that issue by decoupling our meeting from the location, so we change plans and schedule a Google Hangout session. Now we must

- Be available at the same time
- Find each other in Google Hangout

Google Hangout (this also works with all other video/voice chats) does synchronous decoupling. It removes the need to be at the same place while still requiring us to meet at the same time.

We can even decouple from time by using an e-mail conversation. Now we must

- Find each other via email

Email does asynchronous decoupling. You can send an email when the recipient is asleep, and they'll respond when they're awake.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there are no other things going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

NOTE To fully understand this chapter, you'll need to have read and understood the concept of auto-scaling covered in chapter 11.

A meeting isn't the only thing that can be decoupled. In software systems, you can find a lot of tightly coupled components:

- A public IP address is like the location of our meeting. To make a request to a web server, you must know its public IP address, and the server must be connected to that address. If you want to change the public IP address, both parties are involved in making the appropriate changes.
- If you want to make a request to a web server, the web server must be online at the same time. Otherwise your request will fail. There are many reasons a web server can be offline: someone is installing updates, a hardware failure, and so on.

AWS offers a solution for both of these problems. The *Elastic Load Balancing (ELB)* service provides a load balancer that sits between your web servers and the public internet to decouple your servers synchronously. For asynchronous decoupling, AWS offers a *Simple Queue Service (SQS)* that provides a message queue infrastructure. You'll learn about both services in this chapter. Let's start with ELB.

12.1 Synchronous decoupling with load balancers

Exposing a single web server to the outside world introduces a dependency: the public IP address of the EC2 instance. From this point on, you can't change the public IP address again because it's used by many clients sending requests to your server. You're faced with the following issues:

- Changing the public IP address is no longer possible because many clients rely on it.
- If you add an additional server (and IP address) to handle increasing load, it's ignored by all current clients: they're still sending all requests to the public IP address of the first server.

You can solve these issues with a DNS name pointing to your server. But DNS isn't fully under your control. DNS servers cache entries, and sometimes they don't respect your time to live (TTL) settings. A better solution is to use a load balancer.

A load balancer can help to decouple a system where the requester awaits an immediate response. Instead of exposing your web servers to the outside world, you only expose the load balancer to the outside world. The load balancer then redirects requests to the web servers behind it. Figure 12.1 shows how this works.

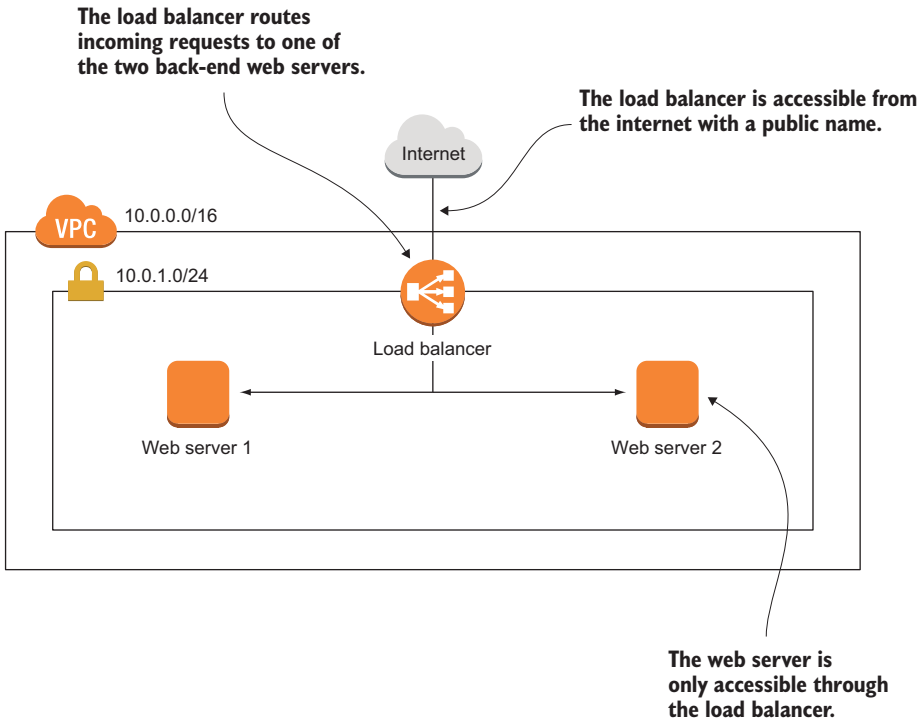


Figure 12.1 A load balancer synchronously decouples your server.

AWS offers load balancers through the ELB service. The AWS load balancer is fault-tolerant and scalable. For each ELB, you pay $ 0.025 per hour and $ 0.008 per GB of processed traffic. The prices are valid for the North Virginia (us-east-1) region.

> **NOTE** The ELB service doesn't have an independent Management Console. It's integrated into the EC2 service.

A load balancer can be used with more than web servers—you can use load balancers in front of any systems that deal with request/response kind of communication.

12.1.1 Setting up a load balancer with virtual servers

AWS shines when it comes to integrating services together. In chapter 11, you learned about auto-scaling groups. You'll now put a Elastic Load Balancer (ELB) in front of an auto-scaling group to decouple traffic to web servers. The auto-scaling group will make sure you always have two servers running. Servers that are started in the auto-scaling group will automatically register with the ELB. Figure 12.2 shows how the setup will look. The interesting part is that the web servers are no longer accessible directly from the public internet. Only the load balancer is accessible and redirects request to the back-end servers behind it; this is done with security groups, which you learned about in chapter 6.

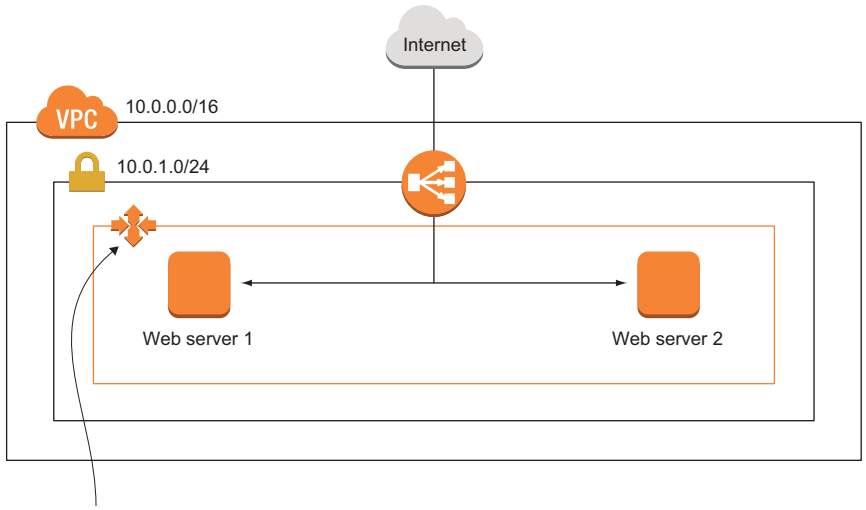


Figure 12.2 Auto-scaling groups work closely with ELB: they register a new server with the load balancer.

An ELB is described by the following:

- The subnets it's attached to. There can be more than one.
- A mapping of the load balancer port to the port on the servers behind the ELB.
- The security groups that are assigned to the ELB. You can restrict traffic to the ELB in the same ways you can with EC2 instances.
- Whether the load balancer should be accessible from the public internet.

The connection between the ELB and the auto-scaling group is made in the auto-scaling group description by specifying `LoadBalancerNames`.

The next listing shows a CloudFormation template snippet to create an ELB and connect it with an auto-scaling group. The listing implements the example shown in figure 12.2.

Listing 12.1 Creating a load balancer and connecting it with an auto-scaling group

```
[...]
"LoadBalancerSecurityGroup": {
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "GroupDescription": "elb-sg",
    "VpcId": {"Ref": "VPC"},
    "SecurityGroupIngress": [{
      "CidrIp": "0.0.0.0/0",
      "FromPort": 80,
      "ToPort": 80,
      "IpProtocol": "tcp"
    }]
  }
},
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "Subnet"}],
    "LoadBalancerName": "elb",
    "Listeners": [{
      "InstancePort": "80",
      "InstanceProtocol": "HTTP",
      "LoadBalancerPort": "80",
      "Protocol": "HTTP"
    }],
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internet-facing"
  }
},
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",
  "Properties": {
    [...]
  }
},
```

The load balancer only accepts traffic on port 80.

Attaches the ELB to the subnet

Maps the load-balancer port to a port on the servers behind it.

Assigns a security group.

The ELB is publicly accessible (use internal instead of internet-facing to define a load balancer reachable from private network only).

```
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",
  "Properties": {
    "LoadBalancerNames": [{"Ref": "LoadBalancer"}],
    "LaunchConfigurationName": {"Ref": "LaunchConfiguration"},
    "MinSize": "2",
    "MaxSize": "2",
    "DesiredCapacity": "2",
    "VPCZoneIdentifier": [{"Ref": "Subnet"}]
  }
}
```

Connects the auto-scaling group with the ELB.

Belongs to MinSize, MaxSize and DesiredCapacity.

To help you explore ELBs, we created a CloudFormation template, located at https://s3.amazonaws.com/awsinaction/chapter12/loadbalancer.json. Create a stack based on that template, and then visit the URL output of your stack with your browser. Every time you reload the page, you should see one of the private IP addresses of a back-end web server.

Cleaning up

Delete the stack you created.

12.1.2 Pitfall: connecting a server too early

The auto-scaling group is responsible for connecting a newly launched EC2 instance with the load balancer. But how does the auto-scaling group knows when the EC2 instance is installed and ready to accept traffic? Unfortunately, the auto-scaling group doesn't know whether the server is ready; it will register the EC2 instance with the load balancer as soon as the instance is launched. If traffic is sent to a server that's launched but not installed, the request will fail and your users will be unhappy.

But the ELB can send periodic health checks to each server that's connected to find out whether the server can serve requests. In the web server example, you want to check whether you get a status code 200 response for a particular resource, such as /index.html. The following listing shows how this can be done with CloudFormation.

Listing 12.2 ELB health checks to determine whether a server can answer requests

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "HealthCheck": {
      "Target": "HTTP:80/index.html",
      "Interval": "10",
      "Timeout": "5",
      "HealthyThreshold": "3",
```

Checks every 10 seconds

Are the servers retuning status code 200 on /index.html?

Timeout after 5 seconds (must be less than Interval)

The check must pass three times in a row to be healthy.

The check must fail two times in a row to be unhealthy.

```
        "UnhealthyThreshold": "2"
      }
    }
  }
```

Instead of /index.html, you can also request a dynamic page like /healthy.php that does some additional checks to decide whether the web server is ready to handle requests. The contract is that you must return an HTTP status code 200 when the server is ready. That's it.

> **Aggressive health checks can cause downtime**
>
> If a server is too busy to answer a health check, the ELB will stop sending traffic to that server. If the situation is caused by a general load increase on your system, the ELB's response will make the situation worse! We've seen applications experience downtime due to overly aggressive health checks. You need proper load testing to understand what's going on. An appropriate solution is application-specific and can't be generalized.

By default, an auto-scaling group determines if an EC2 instance is healthy based on the heath check that EC2 performs every minute. You can configure an auto-scaling group to use the health check of the load balancer instead. The auto-scaling group will now terminate servers not only if the hardware fails, but also if the application fails. Set `"HealthCheckType": "ELB"` in the auto-scaling group description. Sometimes this setting makes sense because restarting can solve issues like memory, thread pool, or disk overflow, but it can also cause unwanted restarts of EC2 instances in the case of a broken application.

12.1.3 More use cases

So far, you've seen the most common use case for ELB: load-balancing incoming web requests to some web servers over HTTP. As mentioned earlier, ELB can do far more than that. In this section, we'll look at four more typical use cases:

1. ELB can balance TCP traffic. You can place almost any application behind a load balancer.
2. ELB can turn SSL-encrypted traffic into plain traffic if you add your SSL certificate to AWS.
3. ELB can log each request. Logs are stored on S3.
4. ELB can distribute your requests evenly across multiple availability zones.

HANDLING TCP TRAFFIC

Until now, you've only used ELB to handle HTTP traffic. You can also configure ELB to redirect plain TCP traffic, to decouple databases or legacy applications with proprietary interfaces. Compared to an ELB configuration that handles HTTP traffic, you

must change `Listeners` and `HealthCheck` to handle TCP traffic with ELB. The health check doesn't check for a specific response as it did when dealing with HTTP; health checks for TCP traffic are healthy when the ELB can open a socket. The following listing shows how you can redirect TCP traffic to MySQL back ends.

Listing 12.3 ELB handling plain TCP traffic (not only HTTP)

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "SubnetA"}, {"Ref": "SubnetB"}],
    "LoadBalancerName": "elb",
    "Listeners": [{
      "InstancePort": "3306",
      "InstanceProtocol": "TCP",
      "LoadBalancerPort": "3306",
      "Protocol": "TCP"
    }],
    "HealthCheck": {
      "Target": "TCP:3306",
      "Interval": "10",
      "Timeout": "5",
      "HealthyThreshold": "3",
      "UnhealthyThreshold": "2"
    },
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internal"
  }
}
```

Redirects traffic on port 3306 (MySQL) to the back-end servers

Healthy when ELB can open a socket on port 3306 on the back-end server

The MySQL database shouldn't be public; choose an internal load balancer.

You can also configure port 80 to be handled as TCP traffic, but you'll lose the ability to do health checks based on the status code that's returned by your web server.

Terminating SSL

An ELB can be used to terminate SSL without the need to do the configuration on your own. Terminating SSL means the ELB offers an SSL-encrypted endpoint that forwards requests unencrypted to your back-end servers. Figure 12.3 shows how this works.

You can use predefined security policies from AWS to get a secure SSL configuration that takes care of SSL vulnerabilities in the wild. You can accept requests on port 443 (HTTPS); the ELB terminates SSL and forwards the request to port 80 on a web server. That's an easy solution to offer SSL-encrypted communication. SSL termination doesn't just work for HTTP requests; it also works for TCP traffic (such as POP3, SMTP, FTP).

> **NOTE** The following example only works if you already own an SSL certificate. If you don't, you need to buy an SSL certificate or skip the example. AWS doesn't offer SSL certificates at the moment. You could use a self-signed certificate for testing purposes.

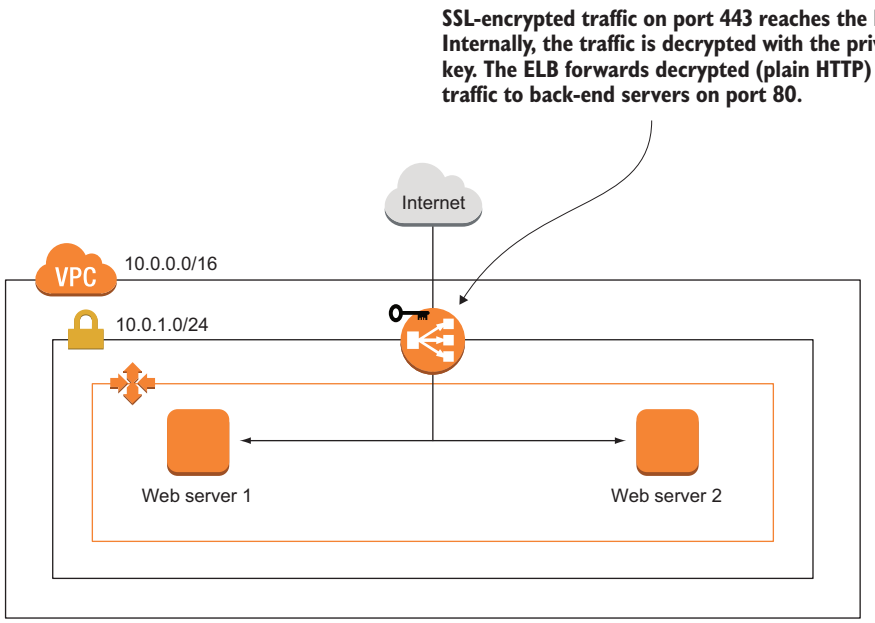


Figure 12.3 A load balancer can accept encrypted traffic, decrypt the traffic, and forward unencrypted traffic to the back end.

Before you can activate SSL encryption, you must upload your SSL certificate to IAM with the help of the CLI:

```
$ aws iam upload-server-certificate \
--server-certificate-name my-ssl-cert \
--certificate-body file://my-certificate.pem \
--private-key file://my-private-key.pem \
--certificate-chain file://my-certificate-chain.pem
```

Now you can use your SSL certificate by referencing `my-ssl-cert`. The following listing shows how encrypted HTTP communication can be configured with the help of the ELB.

Listing 12.4 Terminating SSL with ELB to offer encrypted communication

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "SubnetA"}, {"Ref": "SubnetB"}],
    "LoadBalancerName": "elb",
    "Policies": [{
      "PolicyName": "ELBSecurityPolicyName",
      "PolicyType": "SSLNegotiationPolicyType",
      "Attributes": [{
        "Name": "Reference-Security-Policy",
        "Value": "ELBSecurityPolicy-2015-05"
      }]
```

Configures SSL

Uses a predefined security policy as a configuration

```
    }],
    "Listeners": [{
      "InstancePort": "80",
      "InstanceProtocol": "HTTP",
      "LoadBalancerPort": "443",
      "Protocol": "HTTPS",
      "SSLCertificateId": "my-ssl-cert",
      "PolicyNames": ["ELBSecurityPolicyName"]
    }],
    "HealthCheck": {
      [...]
    },
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internet-facing"
  }
}
```

The back-end servers listen on port 80 (HTTP).

The ELB accepts requests on port 443 (HTTPS).

References the previously uploaded SSL certificate

Terminating SSL with the help of the ELB eliminates many administrative tasks that are critical to providing secure communication. We encourage you to offer HTTPS with the help of an ELB to protect your customers from all kinds of attacks while they're communicating with your servers.

> **WARNING** It's likely that the security policy `ELBSecurityPolicy-2015-05` is no longer the most up-to-date. The security policy defines what versions of SSL are supported, what ciphers are supported, and other security-related options. If you aren't using the latest security policy version, your SSL setup is probably vulnerable. Visit http://mng.bz/916U to get the latest version.

We recommend that you offer only SSL-encrypted communication to your users. In addition to protecting sensitive data, it also has a positive impact on Google rankings.

LOGGING

ELB can integrate with S3 to provide access logs. Access logs contain all the requests processed by the ELB. You may be familiar with access logs from web servers like Apache web server; you can use access logs to debug problems with your back end and analyze how many requests have been made to your system.

To activate access logging, the ELB must know to which S3 bucket logs should be written. You can also specify how often the access logs should be written to S3. You need to set up an S3 bucket policy to allow the ELB to write to the bucket, as shown in the following listing.

Listing 12.5 policy.json

```
{
  "Id": "Policy1429136655940",
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Stmt1429136633762",
```

```
    "Action": ["s3:PutObject"],
    "Effect": "Allow",
    "Resource": "arn:aws:s3:::elb-logging-bucket-$YourName/*",
    "Principal": {
      "AWS": [
        "127311923021", "027434742980", "797873946194",
        "156460612806", "054676820928", "582318560864",
        "114774131450", "783225319266", "507241528517"
      ]
    }
  }]
}
```

To create the S3 bucket with the policy, use the CLI—but don't forget to replace `$YourName` with your name or nickname to prevent name clashes with other readers. This also applies to the policy.json file. To save some time, you can download the policy from https://s3.amazonaws.com/awsinaction/chapter12/policy.json:

```
$ aws s3 mb s3://elb-logging-bucket-$YourName
$ aws s3api put-bucket-policy --bucket elb-logging-bucket-$YourName \
--policy file://policy.json
```

You can activate access logging with the following CloudFormation description.

Listing 12.6 Activating access logs written by ELB

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "AccessLoggingPolicy": {
      "EmitInterval": 10,
      "Enabled": true,
      "S3BucketName": "elb-logging-bucket-$YourName",
      "S3BucketPrefix": "my-application/production"
    }
  }
}
```

How often logs should be written to S3, in minutes (5–60)

Name of the S3 bucket

You can prefix access logs if you want to save multiple access logs to the same S3 bucket (optional).

The ELB will now write access-log files to the specified S3 bucket from time to time. The access log is similar to the one created by the Apache web server, but you can't change the format of the information it contains. The following snippet shows a single line of an access log:

```
2015-06-23T06:40:08.771608Z elb 92.42.224.116:17006 172.31.38.190:80
0.000063 0.000815 0.000024 200 200 0 90
"GET http://elb-....us-east-1.elb.amazonaws.com:80/ HTTP/1.1"
"Mozilla/5.0 (Macintosh; ...) Gecko/20100101 Firefox/38.0" - -
```

Here are examples of the pieces of information an access log always contains:

- Time stamp: `2015-06-23T06:40:08.771608Z`
- Name of the ELB: `elb`

- Client IP address and port: `92.42.224.116:17006`
- Back-end IP address and port: `172.31.38.190:80`
- Number of seconds the request was processed in the load balancer: `0.000063`
- Number of seconds the request was processed in the back end: `0.000815`
- Number of seconds the response was processed in the load balancer: `0.000024`
- HTTP status code returned by the load balancer: `200`
- HTTP status code returned by back end: `200`
- Number of bytes received: `0`
- Number of bytes sent: `90`
- Request: `"GET http://elb-....us-east-1.elb.amazonaws.com:80/ HTTP/1.1"`
- User agent: `"Mozilla/5.0 (Macintosh; ...) Gecko/20100101 Firefox/38.0"`

Cleaning up

Remove the S3 bucket you created in the logging example:

```
$ aws s3 rb --force s3://elb-logging-bucket-$YourName
```

CROSS-ZONE LOAD BALANCING

The ELB is a fault-tolerant service. If you create an ELB, you receive a public name like `elb-1079556024.us-east-1.elb.amazonaws.com` as the endpoint. It's interesting to see what's behind that name. You can use the command-line application `dig` (or `nslookup` on Windows) to ask a DNS server about a particular name:

```
$ dig elb-1079556024.us-east-1.elb.amazonaws.com
[...]
;; ANSWER SECTION:
elb-1079556024.us-east-1.elb.amazonaws.com. 42 IN A 52.0.40.9
elb-1079556024.us-east-1.elb.amazonaws.com. 42 IN A 52.1.152.202
[...]
```

The name `elb-1079556024.us-east-1.elb.amazonaws.com` resolves to two IP addresses: 52.0.40.9 and 52.1.152.202. When you create a load balancer, AWS starts two instances in the background and uses DNS to distributed between the two. To make the servers fault-tolerant, AWS spawns the load-balancer instances in different availability zones. By default, each load-balancer instance of the ELB sends traffic only to EC2 instances in the same availability zone. If you want to distribute requests across availability zones, you can enable *cross-zone load balancing*. Figure 12.4 shows a scenario in which cross-zone load balancing is important.

The following CloudFormation snippet shows how this can be activated:

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "CrossZone": true
  }
}
```

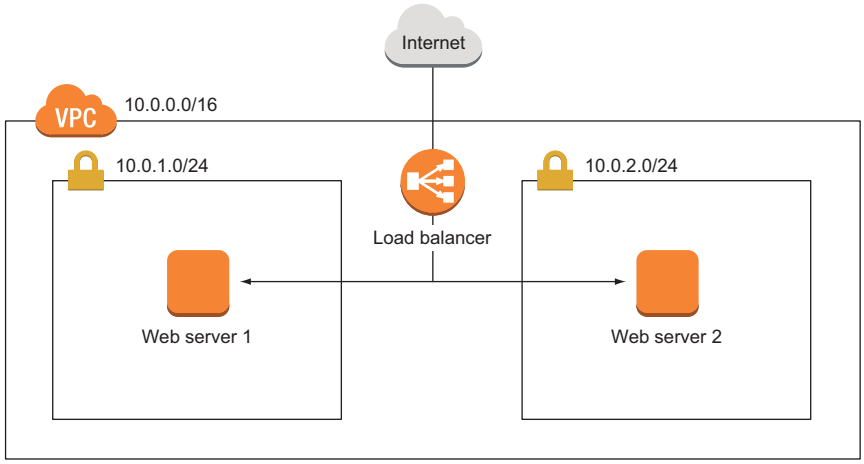


Figure 12.4 Enabling cross-zone load balancing to distribute traffic between availability zones

We recommend that you enable cross-zone load balancing, which is disabled by default, to ensure that requests are routed evenly across all back-end servers.

In the next section, you'll learn more about asynchronous decoupling.

12.2 Asynchronous decoupling with message queues

Synchronous decoupling with ELB is easy; you don't need to change your code to do it. But for asynchronous decoupling, you have to adapt your code to work with a message queue.

A message queue has a head and a tail. You can add new messages to the tail while reading messages from the head. This allows you to decouple the production and consumption of messages. The producers and consumers don't know each other; they both only know about the message queue. Figure 12.5 illustrates this principle.

You can put new messages onto the queue while no one is reading messages, and the message queue acts as a buffer. To prevent message queues from growing infinitely large, messages are only saved for a certain amount of time. If you consume a message from a message queue, you must acknowledge the successful processing of the message to permanently delete it from the queue.

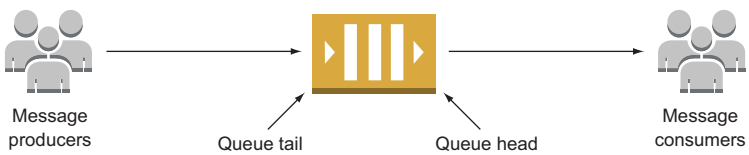


Figure 12.5 Producers send messages to a message queue, and consumers read messages.

The Simple Queue Service (SQS) is a fully managed AWS service. SQS offers message queues that guarantee the delivery of messages at least once:

- Under rare circumstances, a single message will be available for consumption twice. This may sound strange if you compare it to other message queues, but you'll see how to deal with this problem later in the chapter.
- SQS doesn't guarantee the order of messages, so you may read messages in a different order than they were produced.

This limitation of SQS is also beneficial:

- You can put as many messages into SQS as you like.
- The message queue scales with the number of messages you produce and consume.

The pricing model is also simple: you pay $0.00000050 per request to SQS or $0.5 per million requests. Producing a message is one request, and consuming is another request (if your payload is larger than 64 KB, every 64 KB chunk counts as one request).

12.2.1 Turning a synchronous process into an asynchronous one

A typical synchronous process looks like this: a user makes a request to your server, something happens on the server, and a result is returned to the user. To make things more concrete, we'll talk about the process of creating a preview image of an URL in the following example:

1. The user submits a URL.
2. The server downloads the content at the URL and converts it into a PNG image.
3. The server returns the PNG to the user.

With one small trick, this process can be made asynchronous:

1. The user submits a URL.
2. The server puts a message onto a queue that contains a random ID and the URL.
3. The server returns a link to the user where the PNG image will be found in the future. The link contains the random ID (http://$Bucket.s3-website-us-east-1.amazonaws.com/$RandomId.png).
4. In the background, a worker consumes the message from the queue, downloads the content, converts the content into a PNG, and uploads the image to S3.
5. At some point in time, the user tries to download the PNG at the known location.

If you want to make a process asynchronous, you must manage the way the process initiator tracks the status of the process. One way of doing that is to return an ID to the initiator that can be used to look up the process. During the process, the ID is passed from step to step.

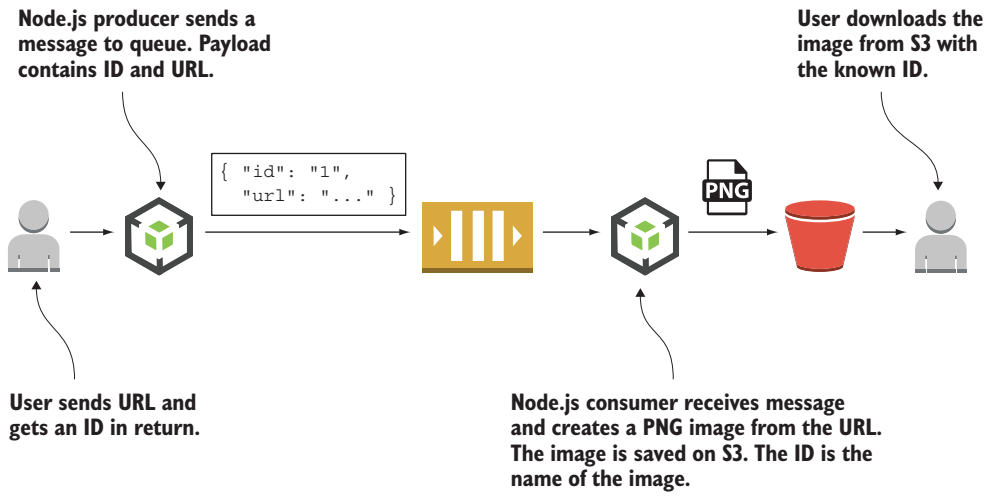


Figure 12.6 How the URL2PNG application works

12.2.2 Architecture of the URL2PNG application

You'll now create a simple but decoupled piece of software named URL2PNG that converts the URL of a web page into a PNG. Again, you'll use Node.js to do the programming part, and you'll use SQS. Figure 12.6 shows how the URL2PNG application works.

To complete the example, you need to create an S3 bucket with web hosting enabled. Execute the following commands, replacing `$YourName` with your name or nickname to prevent name clashes with other readers:

```
$ aws s3 mb s3://url2png-$YourName
$ aws s3 website s3://url2png-$YourName --index-document index.html \
--error-document error.html
```

Web hosting is needed so users can later download the images from S3. Now it's time to create the message queue.

12.2.3 Setting up a message queue

Creating an SQS queue is simple—you only need to specify the name of the queue:

```
$ aws sqs create-queue --queue-name url2png
{
  "QueueUrl": "https://queue.amazonaws.com/878533158213/url2png"
}
```

The returned `QueueUrl` is needed later in the example, so be sure to save it.

12.2.4 Producing messages programmatically

You now have an SQS queue to send messages to. To produce a message, you need to specify the queue and a payload. You'll again use Node.js in combination with the AWS SDK to connect your program with AWS.

Installing and getting started with Node.js

To install Node.js, visit https://nodejs.org and download the package that fits your OS.

Here's how the message is produced with the help of the AWS SDK for Node.js; it will later be consumed by the URL2PNG worker. The Node.js script can then be used like this (don't try to run this command now—you need to install and configure URL2PNG first):

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$YourName.s3-website-us-east-1.amazonaws.com/XYZ.png
```

As usual, you'll find the code in the book's code repository on GitHub https://github.com/AWSinAction/code. The URL2PNG example is located at /chapter12/url2png/. The following listing shows the implementation of index.js.

Listing 12.7 index.js: sending a message to the queue

```
var AWS = require('aws-sdk');
var uuid = require('node-uuid');
var sqs = new AWS.SQS({
  "region": "us-east-1"
});

if (process.argv.length !== 3) {
  console.log('URL missing');
  process.exit(1);
}

var id = uuid.v4();
var body = {
  "id": id,
  "url": process.argv[2]
};

var params = {
  "MessageBody": JSON.stringify(body),
  "QueueUrl": "$QueueUrl"
};

sqs.sendMessage(params, function(err) {
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be available soon at http://url2png-$YourName.s3-
    ➥ website-us-east-1.amazonaws.com/' + id + '.png');
  }
});
```

Creates an SQS endpoint

Checks whether a URL was provided

Creates a random ID

The payload contains the random ID and the URL.

Converts the payload into a JSON string

Queue to which the message is sent (was returned when creating the queue)

Invokes the sendMessage operation on SQS

Before you can run the script, you need to install the Node.js modules. Run `npm install` in your terminal to install the dependencies. You'll find a config.json file that needs to be modified. Make sure to change `QueueUrl` to the queue you created at the beginning of this example and change `Bucket` to `url2png-$YourName`.

Now you can run the script with `node index.js "http://aws.amazon.com"`. The program should response with something like "PNG will be available soon at http://url2png-$YourName.s3-website-us-east-1.amazonaws.com/XYZ.png". To verify that the message is ready for consumption, you can ask the queue how many messages are inside:

```
$ aws sqs get-queue-attributes \
--queue-url $QueueUrl \
--attribute-names ApproximateNumberOfMessages
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}
```

Next, it's time to work on the worker that consumes the message and does all the work of generating a PNG.

12.2.5 Consuming messages programmatically

Processing a message with SQS takes three steps:

1. Receive a message.
2. Process the message.
3. Acknowledge successful message processing.

You'll now implement each of these steps to change a URL into a PNG.

To receive a message from an SQS queue, you must specify the following:

- The queue
- The maximum number of messages you want to receive. To get higher throughput, you can get batches of messages.
- The number of seconds you want to take this message from the queue to process it. Within that time, you must delete the message from the queue, or it will be received again.
- The maximum number of seconds you want to wait to receive messages. Receiving messages from SQS is done by polling the API. But the API allows long-polling for a maximum of 10 seconds.

The next listing shows how this is done with the SDK.

Listing 12.8 worker.js: receiving a message from the queue

```
var fs = require('fs');
var AWS = require('aws-sdk');
var webshot = require('webshot');
```

```
var sqs = new AWS.SQS({
  "region": "us-east-1"
});
var s3 = new AWS.S3({
  "region": "us-east-1"
});

function receive(cb) {
  var params = {
    "QueueUrl": "$QueueUrl",
    "MaxNumberOfMessages": 1,
    "VisibilityTimeout": 120,
    "WaitTimeSeconds": 10
  };
  sqs.receiveMessage(params, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Messages === undefined) {
        cb(null, null);
      } else {
        cb(null, data.Messages[0]);
      }
    }
  });
}
```

Takes the message from the queue for 120 seconds

Consumes not more than one message at once

Long poll for 10 seconds to wait for new messages

Invokes the receiveMessage operation on SQS

Checks whether a message is available

Gets the one and only message

The receive step has now been implemented. The next step is to process the message. Thanks to a Node.js module called `webshot`, it's easy to create a screenshot of a website.

Listing 12.9 worker.js: processing a message (take screenshot and upload to S3)

```
function process(message, cb) {
  var body = JSON.parse(message.Body);
  var file = body.id + '.png';
  webshot(body.url, file, function(err) {
    if (err) {
      cb(err);
    } else {
      fs.readFile(file, function(err, buf) {
        if (err) {
          cb(err);
        } else {
          var params = {
            "Bucket": "url2png-$YourName",
            "Key": file,
            "ACL": "public-read",
            "ContentType": "image/png",
            "Body": buf
          };
          s3.putObject(params, function(err) {
            if (err) {
              cb(err);
```

Creates the screenshot with the webshot module

The message body is a JSON string. You convert it back into a JavaScript object.

Opens the screenshot that was saved to local disk by the webshot module

Allows everyone to read the screenshot on S3

Uploads the screenshot to S3

```
        } else {
          fs.unlink(file, cb);
        }
      });
    }
  });
 }
});
}
```

Removes the screenshot from local disk

The only step that's missing is to acknowledge that the message was successfully consumed. If you receive a message from SQS, you get a `ReceiptHandle`, which is a unique ID that you need to specify when you delete a message from a queue.

Listing 12.10 worker.js: acknowledging a message (deletes the message from the queue)

```
function acknowledge(message, cb) {
  var params = {
    "QueueUrl": "$QueueUrl",
    "ReceiptHandle": message.ReceiptHandle
  };
  sqs.deleteMessage(params, cb);
}
```

ReceiptHandle is unique for each receipt of a message.

Invokes the deleteMessage operation

You have all the parts; now it's time to connect them.

Listing 12.11 worker.js: connecting the parts

```
function run() {
  receive(function(err, message) {
    if (err) {
      throw err;
    } else {
      if (message === null) {
        console.log('nothing to do');
        setTimeout(run, 1000);
      } else {
        console.log('process');
        process(message, function(err) {
          if (err) {
            throw err;
          } else {
            acknowledge(message, function(err) {
              if (err) {
                throw err;
              } else {
                console.log('done');
                setTimeout(run, 1000);
              }
            });
          }
        });
      }
```

Receives a message

Checks whether a message is available

Calls the run method again in one second

Processes the message

Acknowledges the message

Calls the run method again in one second

```
    }
  });
}

run();
```

Calls the run method to start

Now you can start the worker to process the message that is already in the queue. Run the script with `node worker.js`. You should see some output that says the worker is in the process step and that then switches to Done. After a few seconds, the screenshot should be uploaded to S3. Your first asynchronous application is complete.

You've created an application that is asynchronously decoupled. If the URL2PNG service becomes popular and millions of users start using it, the queue will become longer and longer because your worker can't produce that many PNGs from URLs. The cool thing is that you can add as many workers as you like to consume those messages. Instead of only 1 worker, you can start 10 or 100. The other advantage is that if a worker dies for some reason, the message that was in flight will become available for consumption after two minutes and will be picked up by another worker. That's fault-tolerant! If you design your system asynchronously decoupled, it's easy to scale and a good foundation to be fault-tolerant. The next chapter will concentrate on this topic.

Cleaning up

Delete the message queue as follows:

```
$ aws sqs delete-queue --queue-url $QueueUrl
```

And don't forget to clean up and delete the S3 bucket used in the example. Issue the following command, replacing `$YourName` with your name:

```
$ aws s3 rb --force s3://url2png-$YourName
```

12.2.6 Limitations of messaging with SQS

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail.

SQS DOESN'T GUARANTEE THAT A MESSAGE IS DELIVERED ONLY ONCE

If a received message isn't deleted within `VisibilityTimeout`, the message will be received again. This problem can be solved by making the receive idempotent. *Idempotent* means that no matter how often the message is consumed, the result stays the same. In the URL2PNG example, this is true by design: if you process the message multiple times, the same image is uploaded to S3 multiple times. If the image is already available on S3, it's replaced. Idempotence solves many problems in distributed systems that guarantee at least single delivery of messages.

Not everything can be made idempotent. Sending an e-mail is a good example: if you process a message multiple times and it sends an email each time, you'll annoy

the addressee. As a workaround, you can use a database to track whether you already sent the email.

In many cases, at least once is a good trade-off. Check your requirements before using SQS if this trade-off fits your needs.

SQS DOESN'T GUARANTEE THE MESSAGE ORDER

Messages may be consumed in a different order than the order in which you produced them. If you need a strict order, you should search for something else. SQS is a fault-tolerant and scalable message queue. If you need a stable message order, you'll have difficulty finding a solution that scales like SQS. Our advice is to change the design of your system so you no longer need the stable order or produce the order at the client side.

SQS DOESN'T REPLACE A MESSAGE BROKER

SQS isn't a message broker like ActiveMQ—SQS is only a message queue. Don't expect features like those offered by message brokers. Considering SQS versus ActiveMQ is like comparing DynamoDB to MySQL.

12.3 Summary

- Decoupling makes things easier because it reduces dependencies.
- Synchronous decoupling requires two sides to be available at the same time, but the sides don't know each other.
- With asynchronous decoupling, you can communicate without both sides being available.
- Most applications can be synchronously decoupled without touching the code, using the load balancer offered by the Elastic Load Balancing service.
- A load balancer can make periodic health checks to your application to determine whether the back end is ready to serve traffic.
- Asynchronous decoupling is only possible with asynchronous processes. But you can modify a synchronous process to be an asynchronous one most of the time.
- Asynchronous decoupling with SQS requires programming against SQS with one of the SDKs.

The book “Cloud Native Patterns” teaches you how to build software applications that leverage the opportunities of cloud architectures—and avoid their risks. This chapter—“Running Cloud-native Applications in Production”—focuses on designing your applications with principles like continuous delivery, repeatability, and total product life cycle baked in.

Chapter 2 from *Cloud Native Patterns*
by Cornelia Davis

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 efficiencies 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 both

- 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 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] As a result, any instability in the infrastructure propagates up into the running application, making it hard to keep running.
- *Production instability*—And finally, because deploying into an unstable environment is usually inviting more trouble, the frequency of production deployments is limited.

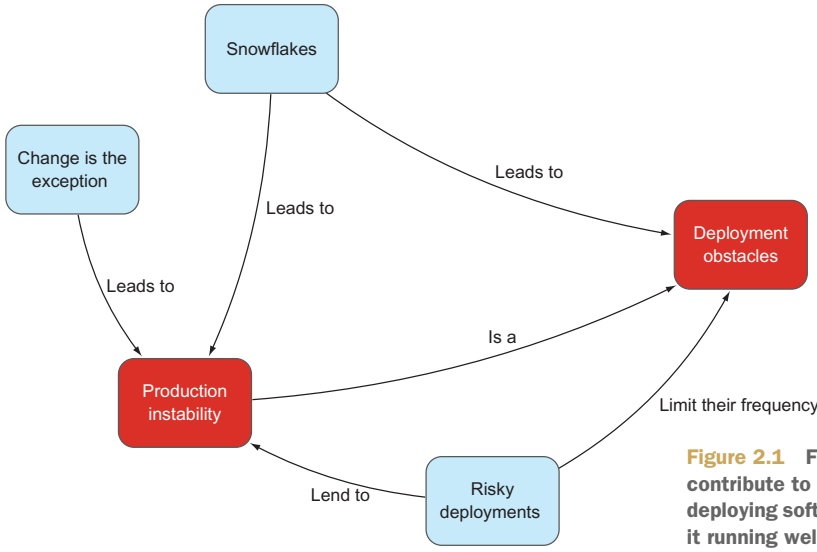


Figure 2.1 Factors that contribute to the difficulty in deploying software and keeping it running well in production

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-,

[1] Wikipedia's "Fallacies of Distributed Computing" entry (https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing) provides more details.

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, the tests are no longer passing, and the 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 actual 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, which is complicated by a multitude of added 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 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.

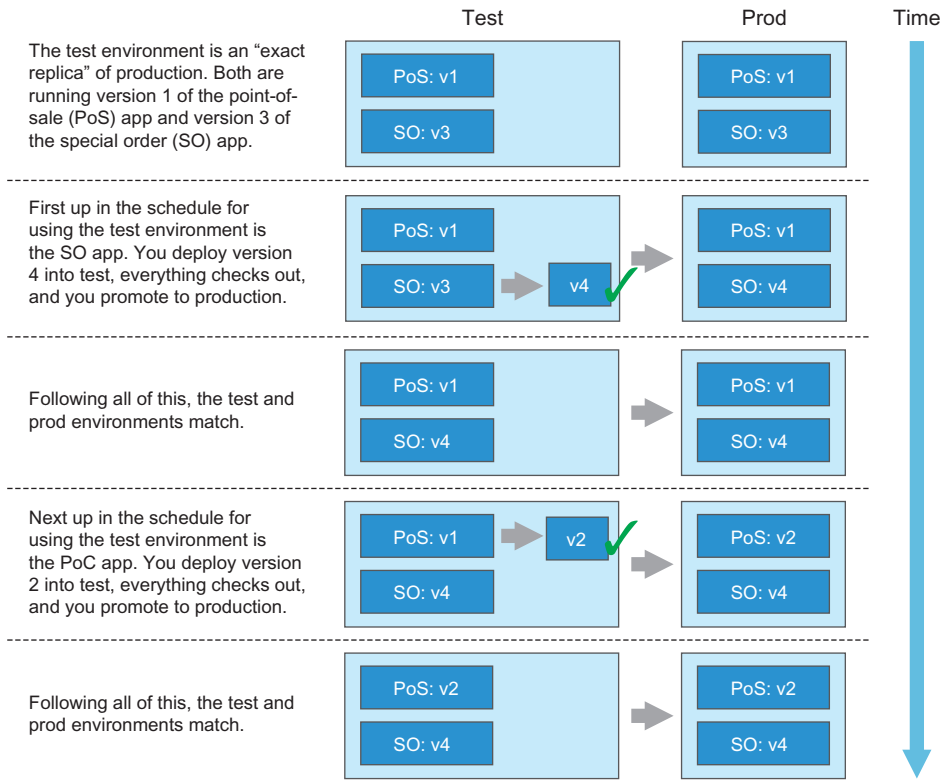


Figure 2.2 Testing two apps in sequence is straightforward when all tests pass.

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 in *test*? Do you revert to version 3 of SO in the test environment (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 shows a couple of alternatives, which get complicated quickly, even in this toy scenario. In a real setting, this becomes intractable.

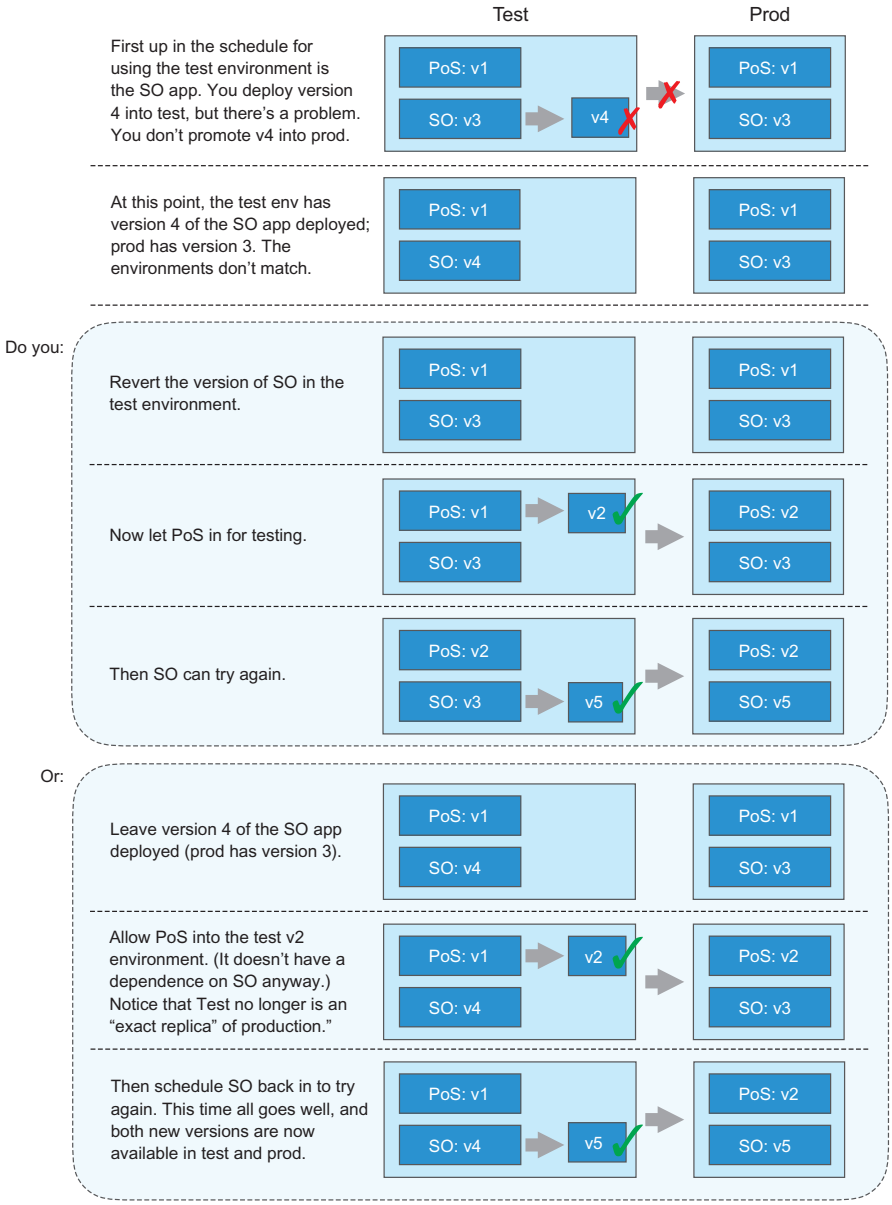


Figure 2.3 A failing test immediately complicates the process for preproduction testing.

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 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 of the factors I've covered until now inarguably help to keep 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 for 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 this using the techniques I just described. A different way exists.

Figure 2.4 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

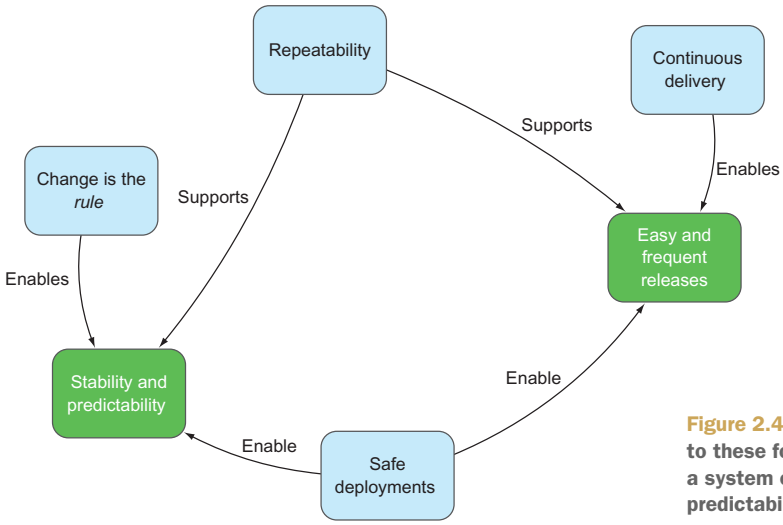


Figure 2.4 Explicit attention to these four factors develops a system of efficiency, predictability, and stability.

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 had 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 this diagram, 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 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 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.

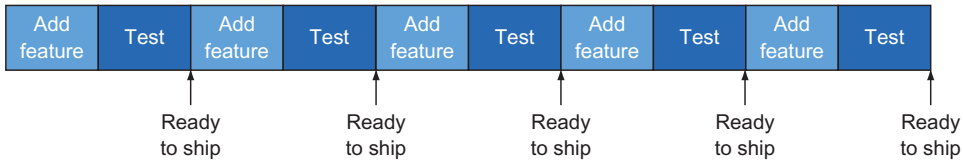


Figure 2.5 Every dev/test cycle doesn't result in a ship; instead, every cycle results in software that's ready to ship. Shipping then becomes a business decision.

Contrast this to the more traditional software development practice depicted in figure 2.6. 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.

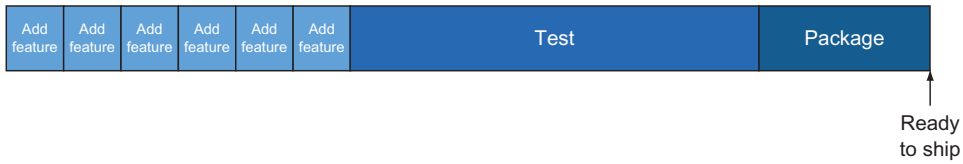


Figure 2.6 A traditional software delivery lifecycle front-loads a lot of development work and a long testing cycle before creating the artifacts that can then be released into production.

Let's assume that the time spans covered by figure 2.5 and figure 2.6 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!" Overlaying that point in time over the previous two diagrams in figure 2.7 shows a stark contrast.

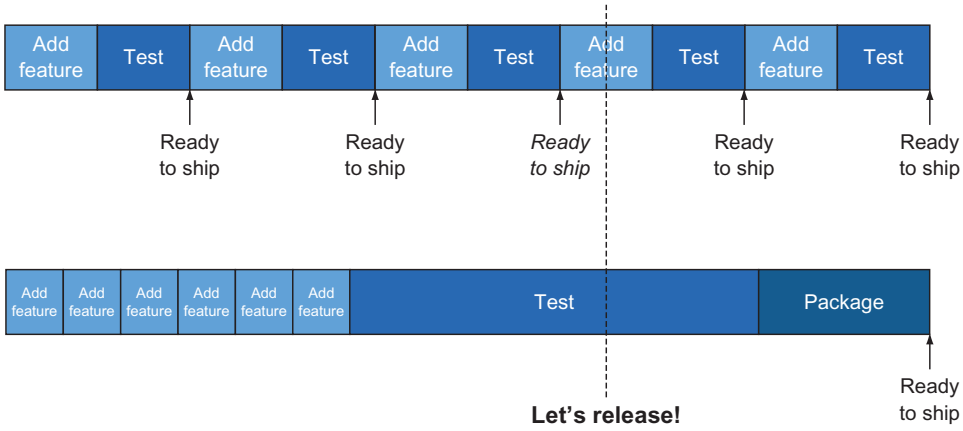


Figure 2.7 **Continuous delivery is concerned with allowing business drivers, not IT readiness, to determine when software is shipped.**

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 diagram, 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 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.

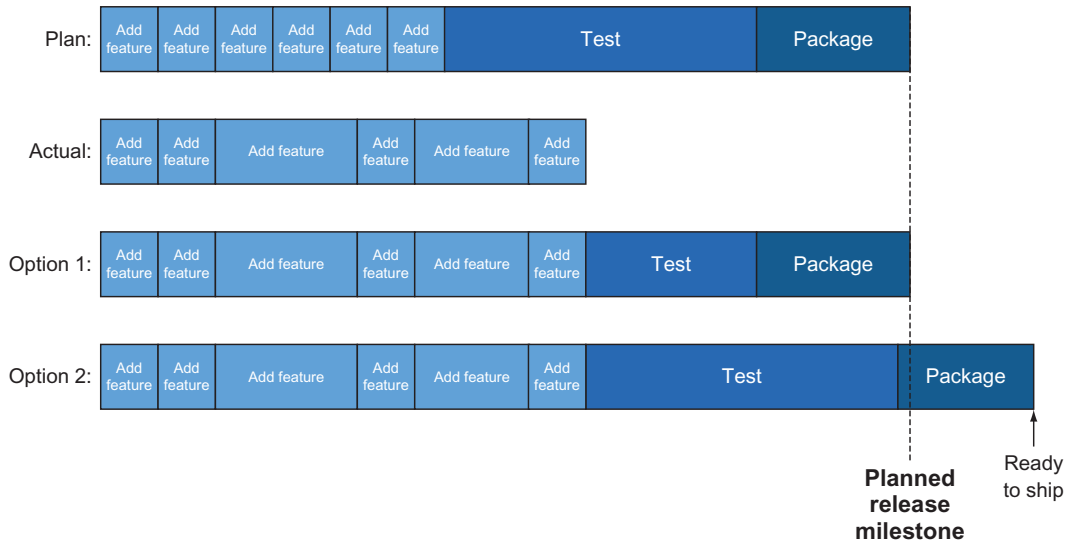


Figure 2.8 When the development schedule slips, you need to decide between two unpalatable options.

Contrast this to the effects that "unanticipated" development delays have on a process that implements many shorter iterations. As depicted in figure 2.9, 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 a 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.

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

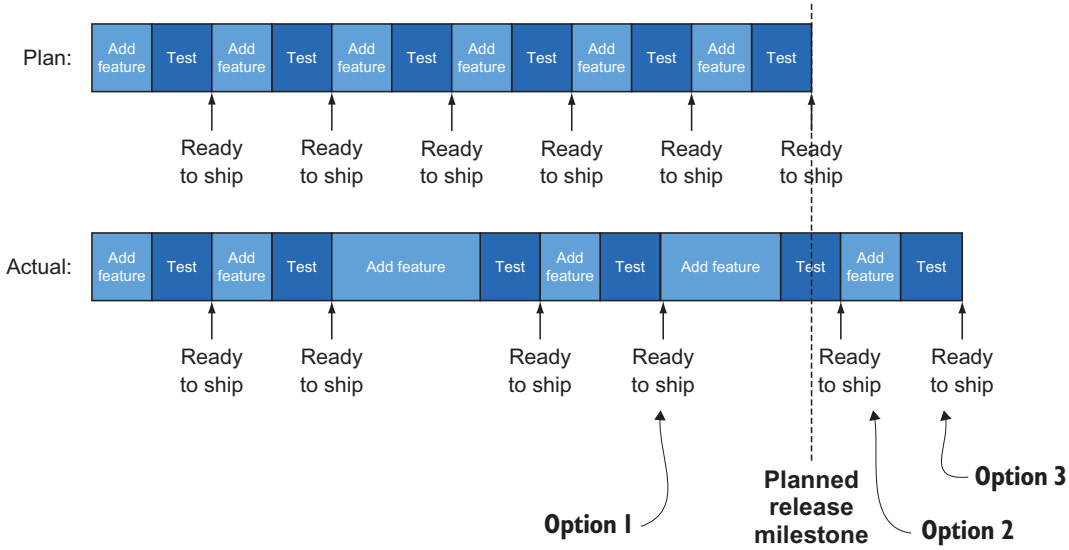


Figure 2.9 **Shorter iterations designed for continuous delivery allow for an agile release process while maintaining software quality.**

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 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 to 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 ¾-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, 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.

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

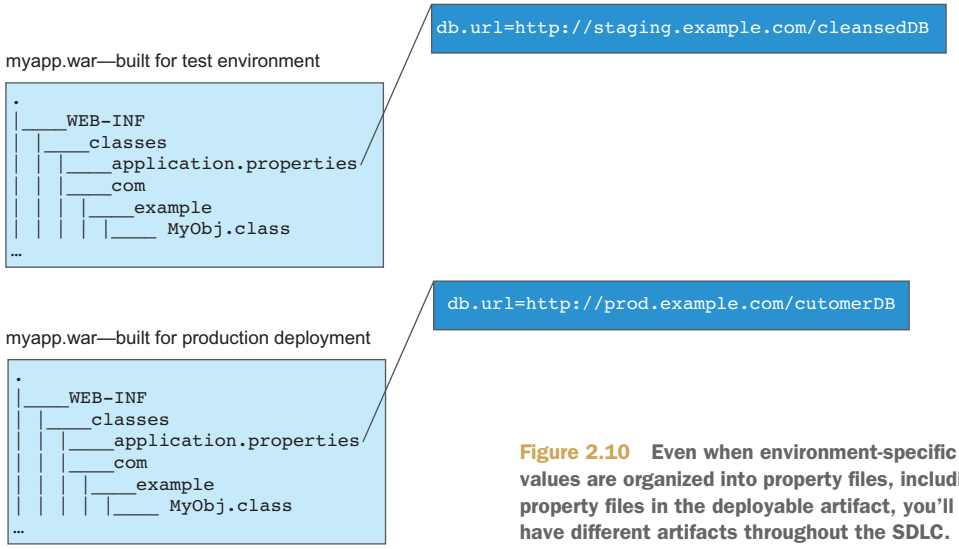


Figure 2.10 **Even when environment-specific values are organized into property files, including property files in the deployable artifact, you'll have different artifacts throughout the SDLC.**

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 depicts the desired outcome: in all stages of the SDLC, you can reliably stamp out exact copies of as many running units as needed.

This picture 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.

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?

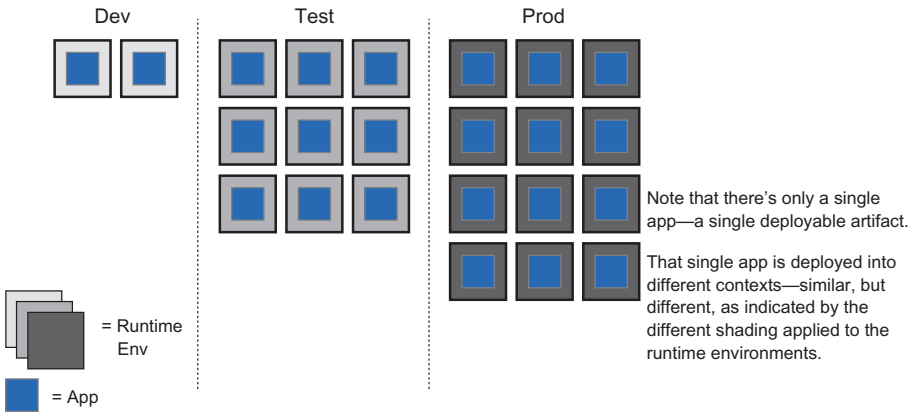


Figure 2.11 **The desired outcome is to be able to consistently establish apps running in standardized environments. Note that the app is the same across all environments; the runtime environment is standardized within an SDLC stage.**

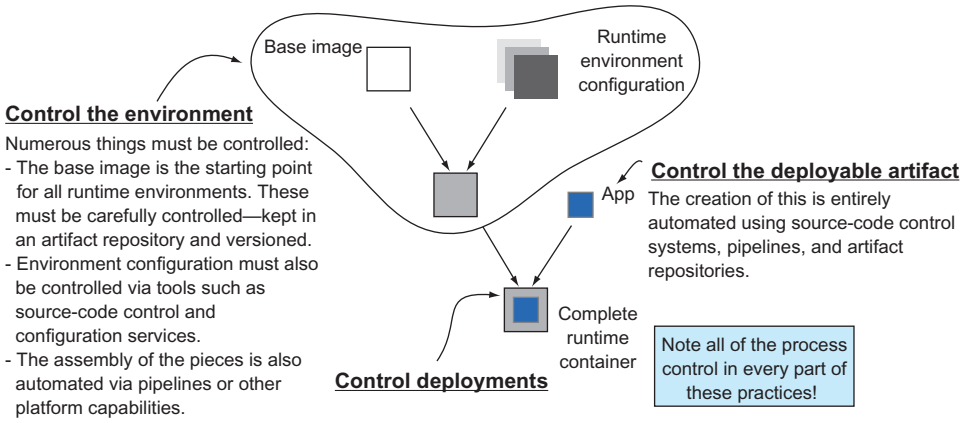


Figure 2.12 **Assembly of standardized base images, controlled environment configurations, and single deployable artifacts are automated.**

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 plat-

form, use an API to download new "stem cells,"[2] 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 functions) happy.

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. Egads! 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 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

[2] See Pivotal's API Documentation page at https://network.pivotal.io/docs/api for more information.

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, 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 then 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 shows how the core practice works.

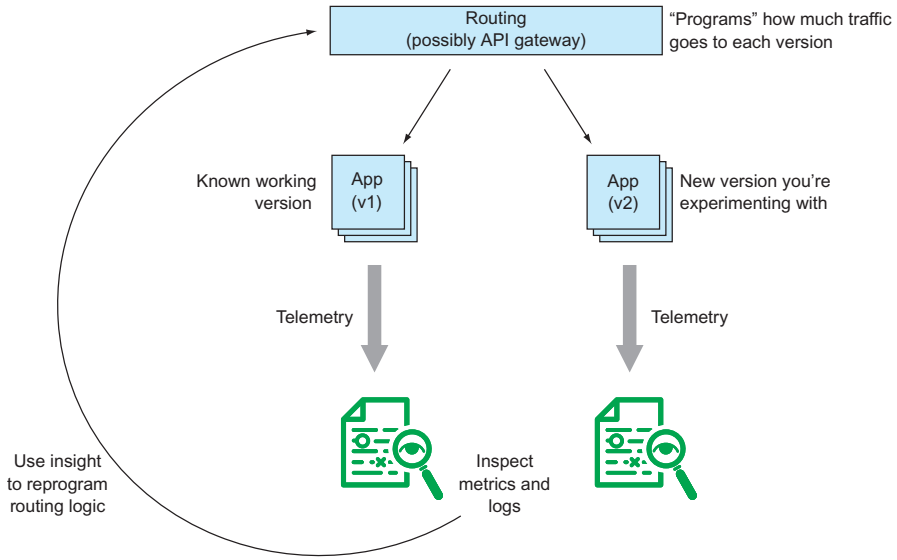


Figure 2.13 Data tells you how parallel deployments of multiple versions of your apps are operating. You use that data to program control flows to those apps, supporting safe rollouts of new software in production.

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]

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), 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 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

[3] Google more details in its "Accelerate: State of DevOps" report, available at https://cloudplatformonline.com/2018-state-of-devops.html.

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, is an expression of the desired state of the system.

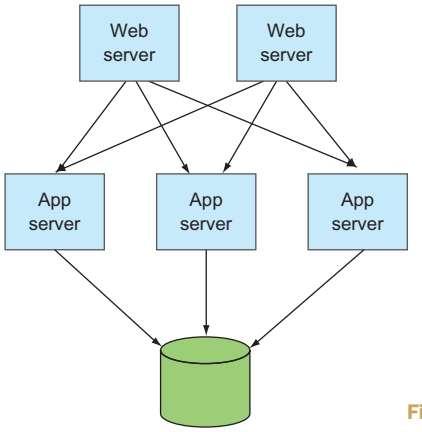


Figure 2.14 The desired state of your deployed software

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 I laid out in the preceding example. 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 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).

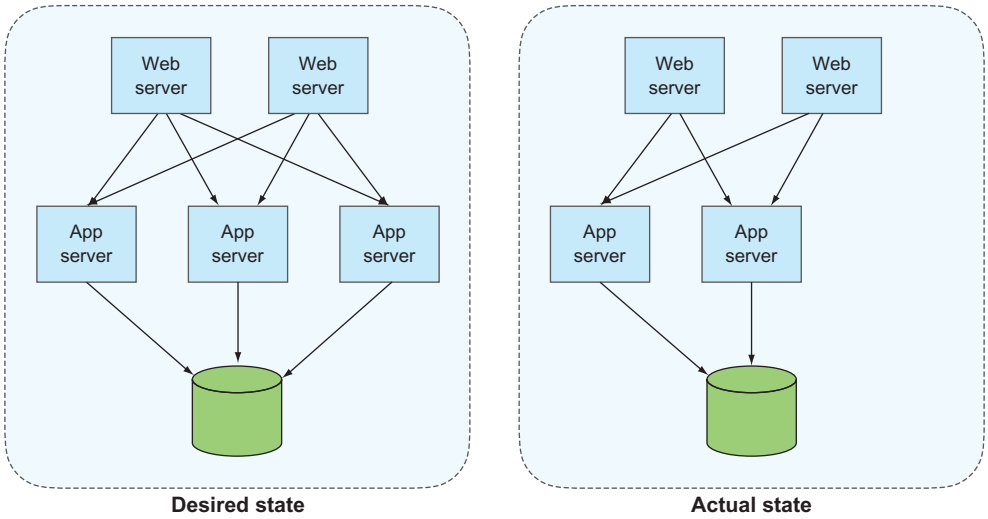


Figure 2.15 **When the actual state doesn't match the desired state, the eventually consistent system initiates actions to bring them back into alignment.**

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 a 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 and further de-risks those deployments. 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.
- That 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.

index

T

U

V

W