O'REILLY®

Second Edition

Architecting for Scale

How to Maintain High Availability and Manage Risk in the Cloud

Free Chapters

compliments of

Cockroach Labs

Lee Atchison

CockroachDB

Build what you dream.

Never worry about your database again.

```
/* Power your apps with the serverless SQL database built for developers.
   Elastic scale, zero operations and free forever. */
```

A hassle-free SQL database

Give your apps a distributed platform that's always available — and free your team from operational headaches.

Cost-effective, elastic and automated scale

Trust in infrastructure that grows alongside your apps. Pay only for what you use, when you use it.

Compatible with PostgreSQL

Build with familiar SQL for transactional consistency and relational schema efficiency.

Start instantly

www.cockroachlabs.com/serverless

TRUSTED BY INNOVATORS

COMCAST

Hard Rock DIGITAL

BOSE

orca security

rubrik

Five9

nubank

SECOND EDITION

Architecting for Scale

How to Maintain High Availability and Manage Risk in the Cloud

This excerpt contains Chapters 1, 4, and 12 of *Architecting for Scale*. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Lee Atchison

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

Architecting for Scale
by Lee Atchison

Printed in the United States of America.

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

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

Acquisitions Editor: Kathleen Carr
Developmental Editor: Amelia Blevins
Production Editor: Beth Kelly
Copyeditor: Jasmine Kwityn
Proofreader: Arthur Johnson

Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2016: First Edition
February 2020: Second Edition

Revision History for the Second Edition
2020-02-28: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492057178* for release details.

978-1-492-05717-8

[LSI]

Table of Contents

Foreword

There was a time in the early days of the internet and app economy when scale was a challenge that only a select few had to fret over. The term "web scale" was coined to address the early challenges of companies like Amazon, Google, Microsoft, and later the architectures of Facebook, Netflix, and others who first reached massive volumes of users. But today, nearly every company sees massive demands on their applications and infrastructure. Even the smallest startup must think about scale from day one.

Modern applications are overwhelmingly data-intensive. We expect highly dynamic, personalized experiences available instantaneously. These applications are increasingly global in nature, as well (think Spotify, Uber, or Square) and have complex needs in trying to balance both consistency and performance. Our applications have become ever more complicated with ever more components that must be managed and scaled.

Further challenging us, users have come to expect absolutely flawless experiences. Consumers have zero tolerance for latency or downtime, even under the heaviest workloads. In this context, your most successful moments—when your application sees rapid, viral growth—have the potential to become your worst day.

This need not be the case. If you plan, test, and continually optimize for scale, then unexpected and rapid growth can come without scrambling to patch or repair a failing stack

CockroachDB is focused on enabling effortless scale and bulletproof resilience and for that reason we are proud to sponsor three important chapters from *Architecting for Scale*: "Understanding, Measuring, and Improving Your Availability," "Services and Data," and "Getting Started Architecting for Scale with the Cloud."

We hope you enjoy this book excerpt, and that you consider CockroachDB to support your next data-intensive app.

CHAPTER 1

Understanding, Measuring, and Improving Your Availability

The Big Game

It's Sunday—the day of the big game. You've invited 20 of your closest friends over to watch the game on your new 300-inch Ultra Max TV. Everyone has come, and your house is full of snacks and beer. Everyone is laughing. The game is about to start. And...

...the lights go out...

...the TV goes dark...

...the game, for you and your friends, is over.

Disappointed, you pick up the phone and call the local power company. The representative, unsympathetically, says: "We're sorry, but we guarantee only 95% availability of our power grid."

Why is availability important? Because your customers expect your service to work... all the time. Anything less than 100% availability can be catastrophic to your business.

No one cares whether your system has great features if they can't use it.

One of the most important topics in architecting for scalable systems is availability. Although there are some companies and some services for which a certain amount of downtime is reasonable and expected, most businesses cannot have any downtime at all without it affecting their customers' satisfaction, and ultimately the company's bottom line.

The following are fundamental questions that all companies must ask as they determine how important system availability is to themselves and their customers. It is these questions, and the inevitable answers to them, that are the core of why availability is critical to highly scaled applications.

Why buy from you?
: Why should someone buy your service if it is not operational when they need it?

What do your customers think?
: What do your customers think or feel when they need to use your service and it's not operational?

How do you make customers happy?
: How can you make your customers happy, make your company money, and meet your business promises and requirements if your service is down?

  Keeping your customers happy and engaged with your system is possible only if your system is operational. There is a direct and meaningful correlation between system availability and customer satisfaction.

High availability is such a critical component of building highly scalable systems that we will devote a significant amount of time to the topic in this book. How do you build a system (a service or application or environment) that is highly available even when a wide range of demands are placed on it?

Availability Versus Reliability

Availability and reliability are two similar yet very different concepts. It is important to understand the difference between them.

Reliability, in our context, generally refers to the quality of a system. Typically, it means the ability of a system to consistently perform according to specifications. You speak of software as reliable if it passes its test suites and does generally what you think it should do. Reliability answers the question:

> "Is the response to my query correct?"

Availability, in our context, generally refers to the ability of your system to perform the tasks it is capable of doing. Is the system around? Is it operational? Is it responding? If the answer is "yes," it is available. Availability answers the questions:

> "Am I getting a response?"
>
> "Did the response arrive in time?"

As you can see, availability and reliability are very similar. It is hard for a system to be available if it is not also reliable, and it is hard for a system to be reliable if it is not also available.

More formally, here is what we mean when we use these terms:

Reliability
: The ability of your system to perform the operations it is intended to perform without making a mistake.

Availability
: The ability of your system to be operational when needed in order to perform those operations.

A system that adds 2 + 3 and gets 6 has poor reliability. A system that adds 2 + 3 and never returns a result at all has poor availability. Reliability can often be fixed by testing. Availability is usually much harder to solve.

You can introduce a software bug in your application that can cause 2 + 3 to produce the answer 6. This can be easily caught and fixed in a test suite.

However, assume you have an application that reliably produces the result 2 + 3 = 5. Now imagine running this application on a computer that has a flaky network connection. The result? You run the application, and sometimes it returns 5, and sometimes it doesn't return anything. The application may be reliable, but it is not available.

We will focus almost exclusively on architecting highly available systems. We will assume your system is reliable, we will assume you know how to build and run test suites, and we will discuss reliability only when it has a direct impact on your system architecture or its availability.

What Causes Poor Availability?

What causes an application that previously performed well to begin exhibiting poor availability? There are many possible causes:

Resource exhaustion
: Increase the number of users or increase the amount of data in use in a system and your application may fall victim to resource exhaustion, resulting in a slower and unresponsive application.

Unplanned load-based changes
: Increases in the popularity of your application might require code and application changes to handle the increased load. These changes, often implemented quickly and at the last minute with little or no forethought or planning, increase the likelihood of problems occurring.

Increased number of moving parts
: As an application gains popularity, it is often necessary to assign more and more developers, designers, testers, and other individuals to work on and maintain it.

This larger number of individuals working on the application creates a large number of moving parts, whether those moving parts are new features, changed features, or just general application maintenance. The more individuals working on the application, the more moving parts within the application and the greater the chance for bad interactions to occur in it.

Outside dependencies
: The more dependencies your application has on external resources, such as SaaS services, infrastructure, or cloud-based services, the more it is exposed to availability problems caused by those resources.

Technical debt
: Increases in the applications complexity typically increase technical debt (i.e., the accumulation of desired software changes and pending bug fixes that often build up over time as an application grows and matures). Technical debt increases the likelihood of a problem occurring.

All fast-growing applications have one, some, or all of these problems. As such, potential availability problems can begin occurring in applications that previously performed flawlessly. The problems can quietly creep up on you, or the problems may start suddenly without warning.

But most growing applications will eventually begin having availability problems.

Availability problems cost you money, they cost your customers money, and they cost you your customers' trust and loyalty. Your company cannot survive for long if you constantly have availability problems.

Building applications designed to scale means building applications designed for high availability.

Measuring Availability

Measuring availability is important to keeping your system highly available. Only by measuring availability can you understand how your application is performing now and examine how your application's availability changes over time.

The most widely held mechanism for measuring the availability of a web application is calculating the percent of time it's accessible for use by customers. We can describe this by using the following formula for a given period:

```
Site availability percentage = total_seconds_in_period – seconds_system_is_down /
total_seconds_in_period
```

Let's consider an example. Suppose that over the month of April, your website was down twice; the first time it was down for 37 minutes, and the second time it was down for 15 minutes. What is the availability of your website?

You can see from the following example that it takes only a small amount of outage to have an impact on your availability percentage:

> Total number of seconds down = (37 + 15) × 60 = 3,120 s
>
> Total number of seconds in month = 30 days × 86,400 s/day = 2,592,000 s
>
> Site availability percentage = total_seconds_in_period – seconds_system_is_down / total_seconds_in_period
>
> Site availability percentage = 2,592,000 s – 3,120 s / 2,592,000 s
>
> Site availability percentage = 99.8795
>
> Your site availability is 99.8795%.

The Nines

Often you will hear availability described as "the nines." This is a shorthand way of indicating high-availability percentages. Table 1-1 illustrates what it means. An application that has "2 nines" availability must be available 99% of the time. This means in a typical month it can be down for 432 minutes and still meet the 99% available goal. By contrast, a "4 nines" application must be available 99.99% of the time, meaning it can be down a mere four minutes in a typical month.

Table 1-1. The nines

Nines	Percentage	Monthly outage
2 nines	99%	432 minutes
3 nines	99.9%	43 minutes
4 nines	99.99%	4 minutes
5 nines	99.999%	26 seconds
6 nines	99.9999%	2.6 seconds

In the preceding example, we see that the website has fallen just short of the 3 nines metric (99.8795% compared to 99.9%). For a website that maintains 5 nines of availability, there can be only 26 *seconds* of downtime every *month*.

What's a reasonable availability number in order to consider your system as high availability? It is impossible to give a single answer to this question because it depends dramatically on your website, your customer expectations, your business needs, and your business expectations. You need to determine for yourself what number is required for your business.

Often, for basic web applications, 3 nines is considered *acceptable availability*. Using Table 1-1, this amounts to 43 minutes of downtime every month.

Planned Outages Are Still Outages

Don't be fooled into thinking your site is highly available when it isn't. Planned and regular maintenance that involves your application being unavailable still count against availability.

Here's a comment that I often overhear: "Our application never fails. That's because we regularly perform system maintenance. By scheduling weekly two-hour maintenance windows and performing maintenance during these windows, we keep our availability high."

Does this group keep its application's availability high?

Let's find out:

> Site availability percentage = total_hours_in_period – hours_system_is_down / total_hours_in_period
>
> hours_in_week = 7 days × 24 hours = 168 hours
>
> hours_unavailable_each_week = 2 hours
>
> Site availability (no failures) = 168 hours – 2 hours / 168 hours = 98.8%
>
> Site availability (no failures) = 98.8%

Without having a single failure of its application, the best this organization can achieve is 98.8% availability. This falls short of even 2 nines availability (98.8% versus 99%).

Planned maintenance hurts nearly as much as unplanned outages. If your customer needs your application to be available and it isn't, your customer has a negative experience. It doesn't matter whether or not you planned for the outage.

Availability by the Numbers

Measuring availability is important to keeping your system highly available, now and in the future. This section discussed a common mechanism for measuring availability and provided some guidelines for what is considered reasonable availability.

Improving Your Availability When It Slips

Your application is operational and online. Your systems are in place, and your team is operating efficiently. Everything seems to be going well. Your traffic is steadily increasing, and your sales organization is very happy. All is well.

Then there's a bit of a slip. Your system suffers an unanticipated outage. But that's OK; your availability has been fantastic until now. A little outage is no big deal. Your traffic is still increasing. Everyone shrugs it off—it was just "one of those things."

Then it happens again—another outage. Oops. Well, OK. Overall, we're still doing well. No need to panic; it was just another "one of those things."

Then another outage...

Now your CEO is a bit concerned. Customers are beginning to ask what's going on. Your sales team is starting to worry.

Then another outage...

Suddenly, your once stable and operational system is becoming less and less stable; your outages are getting more and more attention.

Now you've got real problems.

What happened? Keeping your system highly available is a daunting task. What do you do if availability begins to slip? What do you do if your application availability has fallen or begins to fall, and you need to improve it to keep your customers satisfied?

Knowing what you can do when your availability begins to slip will help you to avoid falling into a vicious cycle of problems. What can you do to avoid your availability slipping? Some key things are:

- Measure and track your current availability
- Automate your manual processes
- Automate your deployment processes
- Maintain and track all configurations in a management system
- Allow quick changes and experiments, with an easy rollback capability if a problem occurs
- Aim to continuously improve your applications and systems
- Keep on top of availability as a core issue as your application changes and grows

The following sections detail these key steps in further detail.

Measure and Track Your Current Availability

To understand what is happening to your availability, you must first measure what your current availability is. Tracking when your application is or is not available gives you an availability percentage that can show how you are performing over a specific period of time. You can use this to determine whether your availability is improving or faltering.

You should continuously monitor your availability percentage and report the results on a regular basis. On top of this, overlay key events in your application, such as

when you performed system changes and improvements. This way you can see whether there is a correlation over time between system events and availability issues. This can help you to identify risks to your availability.

Next, you must understand how your application can be expected to perform from an availability standpoint. A tool that you can use to help manage your application availability is service tiers. These are simply labels associated with services that indicate how critical a service is to the operation of your business. The use of service tiers allows you and your teams to distinguish between mission-critical services and those that are valuable but not essential. We'll discuss service tiers in more depth in Chapter 7.

Finally, create and maintain a *risk matrix*. With this tool, you can gain visibility into the technical debt and associated risk present in your application. Risk matrices are covered more fully in Chapter 9.

Now that you have a way to track your availability and a way of identifying and managing your risk, you will want to review your risk management plans on a regular basis.

Additionally, you should create and implement mitigation plans to reduce your application risks. This will give you a concrete set of tasks you and your development teams can implement to tackle the riskiest parts of your application. This is discussed in detail in Chapter 9.

Automate Your Manual Processes

To maintain high availability, you need to remove unknowns and variables. Performing manual operations is a common way to insert variable results and/or unknown results into your system.

You should never perform a manual operation on a production system.

When you make a change to your system, the change might improve your system, or it might compromise it. Using only repeatable tasks gives you the following:

- The ability to test a task before implementing it. Testing what happens when you make a specific change is critical to avoiding mistakes that cause outages.
- The ability to tweak the task to perform exactly what you want it to do. This lets you implement improvements to the change you are about to make before you actually make the change.
- The ability to have the task reviewed by a third party. This increases the likelihood that the task will have no unexpected side effects.
- The ability to put the task under version control. Version control systems allow you to determine when the task is changed, by whom, and for what reasons.

- The ability to apply the task to related resources. Making a change to a single server that improves how that server works is great. Being able to apply the same change to every affected server in a consistent way makes the task even more useful.
- The ability to have all related resources act consistently. If you continuously make "one-off" changes to resources such as servers, the servers will begin to drift and act differently from one another. This makes it difficult to diagnose problematic servers because there will be no baseline of operational expectation that you can use for comparison.
- The ability to implement repeatable tasks. Repeatable tasks are auditable tasks. Auditable tasks are tasks that you can analyze later for their impact, positive or negative, on the system as a whole.

There are many systems for which no one has access to the production environment. Period. The only access to production is through automated processes and procedures. The owners of these systems lock down their environments like this specifically for the aforementioned reasons.

In summary, if you can't repeat a task, it isn't a useful task. There are many places where adding repeatability to changes will help keep your system and application stable. This includes implementing server configuration changes, making performance-tuning tweaks and adjustments, restarting servers, restarting jobs and tasks, changing routing rules, and upgrading and deploying software packages. We'll now look at some examples of repeatable tasks you should employ.

Automated deploys

By automating deploys, you guarantee that changes are applied consistently throughout your system, and that you can apply similar changes later with known results. Additionally, rollbacks to known good states become more reliable with automated deployment systems.

Configuration management

Rather than "tweaking a configuration variable" in the kernel of a server, use a process to apply the change in an automated manner.

At the very least, write a script that will make the change, and then check that script into your software change management system. This enables you to make the same change to all servers in your system uniformly. Additionally, when you need to add a new server to your system or replace an old one, having a known configuration that can be applied improves the likelihood that you can add the new server to your system safely, with minimal impact.

But even better—and consistent with modern, state of the art, configuration management best practices—is to employ a concept called Infrastructure as Code. Infrastructure as Code involves describing your infrastructure in a standard, machine-readable specification and then passing that specification through an infrastructure tool that will create and/or update your infrastructure and your configuration to match the specification. Tools like Puppet and Chef can help make this process easier to manage.

Then you take this specification and check it into your version control system, so that changes to the specification can be tracked just like code changes can be tracked. Running the specification through the infrastructure tool anytime a change is made to the specification will update your live infrastructure to match the specification.

If anyone needs to make a change to the infrastructure or its configuration, they must make the change to the specification, check the change into version control, and then "deploy" the change via the infrastructure tool to update your live infrastructure to match. In this manner, you can:

1. Ensure all components of the infrastructure have a consistent, known, and stable configuration.
2. Track all changes to the infrastructure so they can be rolled back if needed, or used to assist in correlation with system events and outages.
3. Allow a peer review process, similar to a code review process, to ensure changes to your infrastructure are correct and appropriate.
4. Allow creating duplicate environments to assist in testing, staging, and development with an environment identical to production.

This same sort of process applies to all infrastructure components. This includes not only servers and their operating system configuration but also other cloud components, VPCs, load balancers, switches, routers, network components, and monitoring applications and systems.

For Infrastructure as Code management to be useful, it must be employed for all system changes, all the time. It is never acceptable to bypass the infrastructure management system to make a change, no matter the circumstances. Not ever.

You would be surprised the number of times I have received an operational update email that said something like, "We had a problem with one of our servers last night. We hit a limit to the maximum number of open files the server could handle. So I tweaked the kernel variable and increased the maximum number of open files, and the server is operational again."

That is, it is operating correctly until someone accidentally overwrites the change because there was no documentation of the change. Or until one of the other servers

running the application has the same problem but did not have this change applied to it.

Or someone makes another change, which breaks the application because it is inconsistent with the undocumented change you just made.

Consistency, repeatability, and unfaltering attention to detail are critical to making a configuration management process work. And a standard and repeatable configuration management process such as we describe here is critical to keeping your scaled system highly available.

Change experiments and high frequency changes

Another advantage of having a highly repeatable, highly automated process for making changes and upgrades to your system is that it allows you to experiment with changes. Suppose that you have a configuration change you want to make to your servers that you believe will improve their performance in your application. By using an automated change management process, you can do the following:

- Document your proposed change.
- Review the change with people who are knowledgeable and might be able to provide suggestions and improvements.
- Test the change on servers in a test or staging environment.
- Deploy your change quickly and easily.
- Examine the results quickly. If the change didn't have the desired results, you can quickly roll back to a known good state.

The keys to implementing this process are to have an automated change process with rollback capabilities, and to have the ability to make small changes to your system easily and often.[1] The former lets you make changes consistently; the latter lets you experiment and roll back failed experiments with little to no impact on your customers.

Automated change sanity testing

By having an automated change and deploy process,[2] you can implement an automated sanity test of all changes. You can use a browser testing application for web

1 According to Werner Vogels, CTO of Amazon, in 2014 Amazon did 50 million deploys to individual hosts. That's about one every second.

2 This could be, but does not need to be, a modern continuous integration and continuous deploy (CI/CD) process.

applications or use a synthetic monitoring system to simulate customer interaction with your application.

When you are ready to deploy a change to production, you can have your deployment system first automatically deploy the change to a test or staging environment. You can then have these automated tests run and validate that the changes did not break your application.

If and when those tests pass, you can automatically deploy the change in a consistent manner to your production environment. Depending on how your tests are constructed, you should be able to run the tests regularly on your production environment as well to validate that no changes break anything there.

By automating the entire process, you can increase your confidence that a change will not have a negative impact on your production systems.

Improve Your Systems

Now that you have a system to monitor availability, a way to track risk and mitigations in your system, and a way to easily and safely apply consistent changes to your system, you can focus your efforts on improving the availability of your application itself.

Regularly review your risk matrix and your recovery plans. Make reviewing them part of your postmortem process. Execute projects that are designed to mitigate the risks identified in your matrix. Roll out those changes in an automated and safe way, using the sanity tests discussed earlier. Examine how the mitigation has improved your availability. Continue the process until your availability reaches the level you want and need it to be.

Keep on Top of Availability in Your Changing and Growing Application

As your system grows, you'll need to handle larger and larger traffic and data demands. Much of the content in this book is designed to help you address application availability and scalability issues as your application grows and changes. In particular, managing mistakes and errors at scale is discussed in Chapter 2. Service tiers, which you can use to identify key availability-impacting services, are discussed in Chapter 7. And service-level agreement (SLA) management is discussed in Chapter 8.

Typically, your application will change continuously. As such, your risks, mitigations, contingencies, and recovery plans need to constantly change.

Knowing what you can do when your availability begins to slip will help you to avoid falling into a vicious cycle of problems.

Five Focuses to Improve Application Availability

Building a scalable application that has high availability is not easy and does not come automatically. Problems can crop up in unexpected ways that can cause your beautifully functioning application to stop working for all or some of your customers.

These availability problems often arise from the areas you least expect, and some of the most serious availability problems can originate from extremely benign sources.

A Simple Icon Failure

A classic example of the pitfalls of ignoring dependency failure occurred in a real-life application I worked on. The application provided a service to customers, and on the top of every page was a customizable icon representing the currently logged-in user. The icon was generated by a third-party system.

One day, the third-party system that generated the icon failed. Our application, which assumed that system would always work, didn't know what to do. As a result, our application failed as well. Our entire application failed simply because the icon generation system—a very minor "feature"—failed.

How could we have avoided this problem? If we had simply anticipated that the third-party system might fail, we would have walked through this failure scenario during design and discovered that our application would fail subsequently. We could then have added logic to detect the failure and remove the icon if the failure occurred, or simply catch the error when it occurred and not allowed it to propagate down and affect the working aspects of the page.

A simple check and some error recovery logic would have kept the application operational. Instead, our application experienced a major site outage.

All because of the lack of an icon.

No one can anticipate where problems will come from, and no amount of testing will find all issues. Many of these are systemic problems, not merely code problems.

To find these availability problems, we need to step back and take a systemic look at your application and how it works. Here are five things you can and should focus on when building a system to make sure that, as its use scales upwards, availability remains high:

- Build with failure in mind
- Always think about scaling
- Mitigate risk

- Monitor availability
- Respond to availability issues in a predictable and defined way

Let's look at each of these individually.

Focus #1: Build with Failure in Mind

As Werner Vogels, CTO of Amazon, says, "Everything fails all the time." Plan on your applications and services failing. It will happen. Now, deal with it.

Assuming your application will fail, how will it fail? As you build your system, consider availability concerns during all aspects of your system design and construction.

Design

What design constructs and patterns have you considered or are you using that will help improve the availability of your software?

Using design constructs and patterns, such as simple error catching deep within your application, retry logic, and circuit breakers, allows you to catch errors when they have affected the smallest available subset of functionality. This allows you to limit the scope of a problem and have your application still provide useful capabilities even if part of the application is failing.

Dependencies

What do you do when a component you depend on fails? How do you retry? What do you do if the problem is an unrecoverable (hard) failure, rather than a recoverable (soft) failure?

Circuit breaker patterns are particularly useful for handling dependency failures because they can reduce the impact a dependency failure has on your system. Without a circuit breaker, you can decrease the performance of your application because of a dependency failure (e.g., because an unacceptably long timeout is required to detect the failure). With a circuit breaker, you can "give up" and stop using a dependency until you are certain that dependency has recovered.

Customers

What do you do when a component that is a customer of your system behaves poorly? Can you handle excessive load on your system? Can you throttle excessive traffic? Can you handle garbage data passed in? What about excessive data?

Sometimes denial-of-service attacks can come from "friendly" sources. For example, a customer of your application may see a sudden surge in activity that requires a significant increase in the volume of requests to your application. Alternatively, a bug in your customer's application may cause them to call your application at an

unacceptably high rate. What do you do when this happens? Does the sudden increase in traffic bring your application down? Or can you detect this problem and throttle the request rate, limiting or removing the impact to your application?

Focus #2: Always Think About Scaling

Just because your application works now does not mean it will work tomorrow. Most web applications have increasing traffic patterns. A website that generates a certain amount of traffic today might generate significantly more traffic sooner than you anticipate. As you build your system, don't build it for today's traffic; build it for tomorrow's traffic.

Specifically, this might mean:

- Architect in the ability to increase the size and capacity of your databases.
- Think about what logical limits exist to your data scaling. What happens when your database tops out in its capabilities? Identify and remove these limits before your usage approaches them.
- Build your application so that you can add additional application servers easily. This often involves being observant of where and how state is maintained and of how traffic is routed.
- Redirect static traffic to offline providers. This allows your system to deal only with the dynamic traffic that it is designed to deal with. Using external content delivery networks (CDNs) not only can reduce the traffic your network has to handle but also allows the efficiencies of scale that CDNs provide to get that static content to your customers more quickly.
- Think about whether specific pieces of dynamic content can actually be generated statically. Often, content that appears dynamic is actually mostly static, and the scalability of your application can be increased by making this content static. This "dynamic that can be static" data is sometimes hidden where you don't expect it.

Is It Static, or Is It Dynamic?

Often, content that seems dynamic is actually mostly static. Think about a typical top banner on a simple website. Frequently, this content is mostly static, but occasionally there is some dynamic content included in it. For example, the top of the page might say "Log in" if you are not logged in, and "Hello, Lee" if you are logged in (assuming your name is Lee).

Does that mean the entire page must be generated dynamically? Not necessarily. With the exception of the login/greeting portion of the page, the page (or page portion) is static and can easily be provided by a CDN without any computation on your part.

When the majority of the banner is static, you can, in the user's browser, add the changeable content to the page dynamically (such as adding "Log in" or "Hello, Lee" as appropriate). By grouping this dynamic data together and processing it separately from the truly static data, you can increase the performance of your web page and decrease the amount of dynamic work your application has to perform. This increases scalability and, ultimately, availability.

Focus #3: Mitigate Risk

Keeping a system highly available requires removing risk from the system. Often the cause of a system failure could have been identified as a risk before the failure actually occurred. Identifying risk is a key method of increasing availability. All systems have risk in them. There is risk that:

- A server will crash
- A database will become corrupted
- A returned answer will be incorrect
- A network connection will fail
- A newly deployed piece of software will fail

Keeping a system available requires removing risk. But as systems become more and more complicated, this becomes less and less possible. Keeping a large system available is more about managing what your risk is, how much risk is acceptable, and what you can do to mitigate that risk.

We call this risk management. We will be talking extensively about risk management in Chapter 9. Risk management is at the heart of building highly available systems.

Part of risk management is risk mitigation. Risk mitigation is knowing what to do when a problem occurs in order to reduce the impact of the problem as much as possible. Mitigation is about making sure your application works as well and as

completely as possible, even when services and resources fail. Risk mitigation requires thinking about the things that can go wrong and putting a plan together *now* so that you will be able to handle the situation when it does happen.

Risk Mitigation: The No-Search Web Store

Imagine a web store that sells T-shirts. It's your typical online store that provides the ability to browse shirts on a home page, navigate to browse different categories of shirts, and search for a specific style or type of shirt.

To implement the search capability, a store such as this typically needs to invoke a separate search engine, which may be a separate service or may even be a third-party search provider.

However, because the search capability is an independent capability, there is risk to your application that the search service will not be able to function. Your risk management plan identifies this issue and lists "Failed Search Engine" as a risk to your application.

Without a risk mitigation plan, a failed search service might simply generate an error page or perhaps generate incorrect or invalid results—in either case, it is a bad customer experience.

A risk mitigation plan for this example may say something like this:

> We know that our most popular T-shirts are our red-striped T-shirts; 60 percent of people who search our site end up looking at (and hopefully eventually buy) our famous red-striped shirts. So if our search service stops functioning, we will show an "I'm Sorry" page, followed by a list of our most popular T-shirts, including our red-striped shirts. This will encourage customers who encounter this error page to continue to browse to shirts customers have historically found as interesting.
>
> Additionally, we will show a "10% off next purchase" coupon, so that customers who can't find what they are looking for will be enticed to come back to our site in the future when our search service is functional again, rather than looking elsewhere.

The preceding sidebar is an example of risk mitigation; the process of identifying the risk, determining what to do, and implementing these mitigations is risk management.

This process will often uncover unknown problems in your application that you will want to fix immediately instead of waiting for them to occur. It also can create processes and procedures to handle known failure modes so that the cost of that failure is reduced in duration or severity.

Availability and risk management go hand in hand. Building a highly available system is significantly about managing risk.

Focus #4: Monitor Availability

You can't know if there is a problem in your application unless you can see a problem. Make sure your application is properly instrumented so that you can see how the application is performing from an external perspective as well as by way of internal monitoring.

Proper monitoring depends on the specifics of your application and needs, but it usually entails some of the following capabilities:

Server monitoring
: To monitor the health of your servers and make sure they keep operating efficiently.

Configuration change monitoring
: To monitor your system configuration and identify if and when changes to your infrastructure impact your application.

Application performance monitoring
: To look inside your application and services to make sure they are operating as expected.

Synthetic testing
: To examine in real time how your application is functioning from the perspective of your users, in order to catch problems customers might see before they actually see them.

Alerting
: To inform appropriate personnel when a problem occurs so that it can be quickly and efficiently resolved, minimizing the impact on your customers.

There are many good monitoring systems available, both free and paid services. I personally recommend New Relic. It provides all of the aforementioned monitoring and alerting capabilities. As a Software as a Service (SaaS) offering, it can support your monitoring needs at pretty much any scale your application may require.

After you have started monitoring your application and services, start looking for trends in your performance. When you have identified the trends, you can look for outliers and treat them as potential availability issues. You can use these outliers by having your monitoring tools send you an alert when they are identified, before your application fails. Additionally, you can track as your system grows and make sure your scalability plan will continue to work.

Establish internal, private operational goals for service-to-service communications, and monitor them continuously. This way, when you see a performance- or availability-related problem, you can quickly diagnose which service or system is responsible and address the problem. Additionally, you can see "hot spots"—areas in

which your performance is not what it could be—and put development plans in place to address these issues.

Focus #5: Respond to Availability Issues in a Predictable and Defined Way

Monitoring systems are useless unless you are prepared to act on the issues that arise. This means being alerted when problems occur so that you can take action. Additionally, you should establish processes and procedures that your team can follow to help diagnose issues and easily fix common failure scenarios.

For example, if a service becomes unresponsive, you might have a set of remedies to try to make the service responsive. This might include tasks such as running a test to help diagnose where the problem is, restarting a daemon that is known to cause the service to become unresponsive, or rebooting a server if all else fails. Having standard processes in place for handling common failure scenarios will decrease the amount of time your system is unavailable. Additionally, these processes can provide useful follow-up diagnostic information to your engineering teams to help them deduce the root cause of common ailments.

When an alert is triggered for a service, the owners of that service must be the first ones alerted. They are, after all, the ones responsible for fixing any issues with their service. However, other teams who are closely connected to the troubled service and depend on it might also want to be alerted of problems when they occur. For example, if a team makes use of a particular service, they may want to know when that service fails so that they can potentially be more proactive in keeping their systems active during the dependent service outage.

These standard processes and procedures should be part of an online support manual available to all team members who handle on-call responsibility. These support artifacts should also contain contact lists for owners of related services and systems as well as contacts to call to escalate the problem if a simple solution is not possible. There are SaaS applications available that can automate the management and versioning of these support documents and make them available on demand during events.

All of these processes, procedures, and support manuals should be prepared ahead of time so that during an outage your on-call personnel know exactly what to do in various circumstances to restore operations quickly. These processes and procedures are especially useful because outages often occur during inconvenient times, such as the middle of the night or weekends—times when your on-call team might not perform at peak mental efficiency. These recommendations will assist your team in making smarter and safer moves toward restoring your system to operational status.

Being Prepared

No one can anticipate where and when availability issues will occur. But you can assume that they will occur, especially as your system scales to larger customer demands and more complex applications. Being prepared in advance to handle availability concerns is the best way to reduce the likelihood and severity of problems. The information in this chapter, including the five focuses, offers a solid strategy for keeping your applications highly available.

CHAPTER 4

Services and Data

As you build and migrate your application to a service-based architecture, it is critically important to be mindful of where you store data and state within your application.

Stateless Services—Services Without Data

Stateless services are services that manage no data and no state of their own. The entire state and all data that the service requires to perform its actions is passed in (or referenced) in the request sent to the service.

Stateless services offer a huge advantage for scaling. Because they are stateless, it is usually an easy matter to add additional server capacity to a service in order to scale it to a larger capacity, both vertically and horizontally. You get maximum flexibility in how and when you can scale your service if your service does not maintain state.

Additionally, certain caching techniques on the frontend of the service become possible if the cache does not need to concern itself with service state. This caching lets you handle higher scaling requirements with fewer resources.

Not all services can be made stateless, obviously, but for those services that can be stateless, it is a huge advantage for scalability.

Stateful Services—Services with Data

When you do need to store data, given what we just discussed in the preceding section, it might seem obvious to store data in as few services and systems as possible. It might make sense to keep all of your data close to one another to reduce the footprint of the services that need to know and manage your data.

Nothing could be further from the truth.

Instead, localize your data as much as possible. Have services and data stores manage only the data they need to perform their jobs. Other data should be stored in different servers and data stores, closer to the services that require that data.

Localizing data this way provides a few benefits:

Reduced size of individual datasets
: Because your data is split across datasets, each dataset is smaller in size. Smaller dataset size means reduced interaction with the data, making scalability of the database easier. This is called functional partitioning. You are splitting your data based on functional lines rather than on the size of the dataset.

Localized access
: Frequently when you access data in a database or data store, you are accessing all the data within a given record or set of records. Often, much of that data is not needed for a given interaction. By using multiple reduced dataset sizes, you reduce the amount of unneeded data from your queries.

Optimized access methods
: By splitting your data into different datasets, you can optimize the type of data store appropriate for each dataset. Does a particular dataset need a relational data store? Or is a simple key/value data store acceptable?

Keeping your data associated with the services that consume the data will create a more scalable solution, and easier-to-manage architecture and will allow your data requirements to more easily expand as your application expands.

Data Partitioning

Data partitioning can mean many things. In this context, it means partitioning data of a given type into segments based on some key or identifier within the data. It is often done to make use of multiple databases to store larger datasets or datasets accessed at a higher frequency than a single database can handle.

There are other types of data partitioning (such as the aforementioned functional partitioning); however, in this section, we are going to focus on this key-based partitioning scheme.

A simple example of data partitioning is to partition all data for an application by account, so that all data for accounts whose name begins with A–D is in one database, all data for accounts whose name begins with E–K is in another database, and so on

(see Figure 4-1).[1] This is a very simplistic example, but data partitioning is a common tool used by application developers to dramatically scale the number of users who can access the application at any one time, as well as to scale the size of the dataset itself.

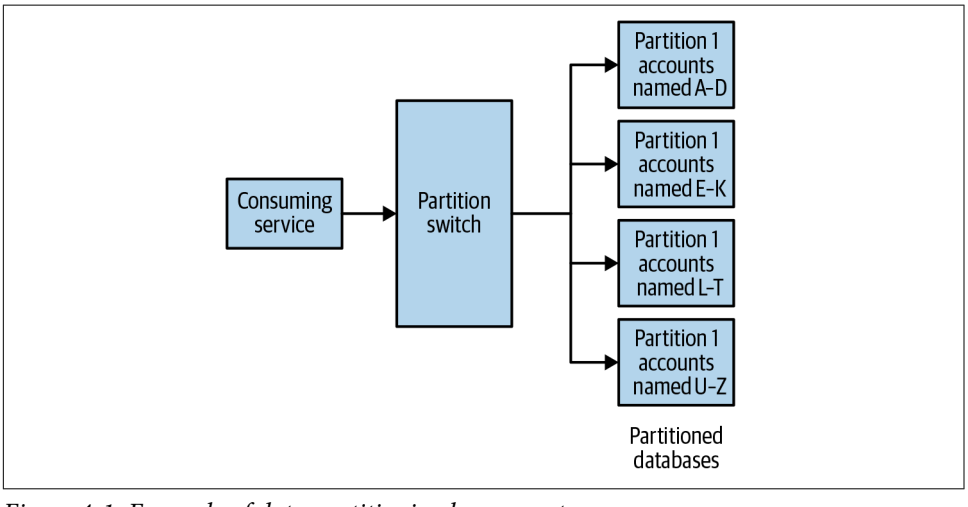

Figure 4-1. Example of data partitioning by account name

In general, you should avoid data partitioning whenever possible. Why? Well, whenever you partition data this way, you run into several potential issues:

Application complexity
: You increase the complexity of your application because you now have to determine where your data is stored before you can actually retrieve it.

Cross-partition queries
: You remove the ability to easily query data across multiple partitions. This is specifically useful in doing business analysis queries.

Skewed partition usage
: Choosing your partitioning key carefully is critical. If you choose the wrong key, you can skew the usage of your database partitions, making some partitions run hotter and others colder, thus reducing the effectiveness of the partitioning while complicating your database management and maintenance. This is illustrated in Figure 4-2.

1 A more likely account-based partitioning mechanism would be to partition by an account identifier rather than by account name. However, using account name makes this example easier to follow.

Repartitioning

Repartitioning is occasionally necessary to balance traffic across partitions effectively. Depending on the key chosen and the type and size of the dataset, this can prove to be an extremely difficult task, an extremely dangerous task (data migration), and in some cases, a nearly impossible task.

In general, account name or account ID is almost always a bad partition key (yet it is one of the most common keys chosen). This is because a single account can change in size during the life of that account. Take a look at Figure 4-2. An account might begin small and thus may easily fit on a partition with a significant number of small accounts. However, if it grows over time, it can soon cause that single partition to not be able to handle all of the load appropriately, and you'll need to repartition in order to better balance account usage. If a single account grows too large, it can actually be bigger than what can fit on a single partition, which will make your entire partitioning scheme fail, because no rebalancing will solve that problem.

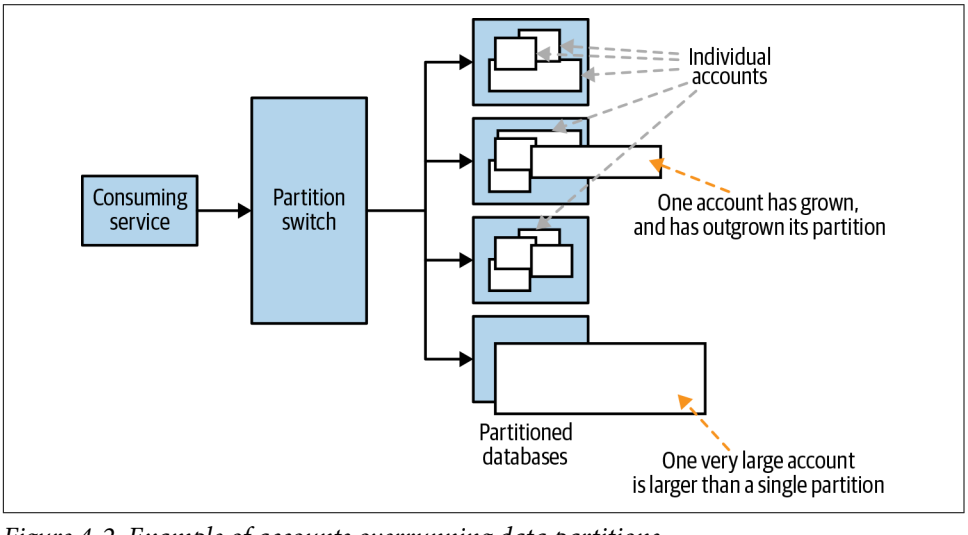

Figure 4-2. Example of accounts overrunning data partitions

A better partition key would be one that would result in consistently sized partitions as much as possible. Growth of partitions should be as independent and consistent as possible, as shown in Figure 4-3. If repartitioning is needed, it should be because all partitions have grown consistently and are too big to be handled by the database.

One potentially useful partitioning scheme is to use a key that generates a significant number of small elements. Next, map these small partitions onto larger partitioned databases. Then, if repartitioning is needed, you can simply update the mapping and move individual small elements to new partitions, removing the need for a massive

repartitioning of the entire system. Selecting and utilizing appropriate partition keys is an art in and of itself.

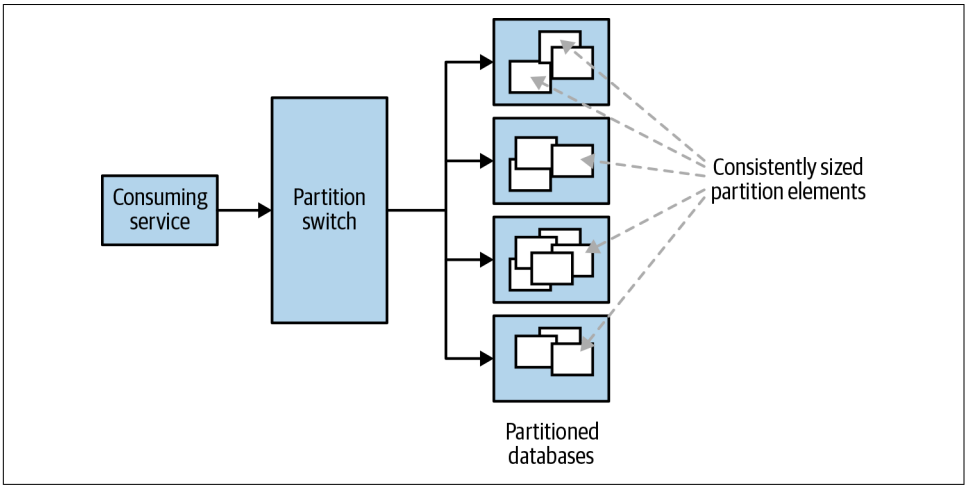

Figure 4-3. Example of consistently sized partitioned elements

Timely Handling of Growing Pains

Most modern applications experience growth in their traffic requirements, in the size and complexity of the applications themselves, and in the number of people working on the applications. Often, we ignore these growing pains, waiting until the pain reaches a certain threshold before we attempt to deal with it. However, by that point, it is usually too late. The pain has reached a serious level, and many easy techniques to help reduce it are no longer available for you to use.

If we don't think about how our application may grow while we are architecting the application before it scales, we will lock ourselves into architectural decisions that can block our ability to scale as our business requires.

Instead, while designing and architecting your new application and changes to your existing applications, consider how those changes will be impacted by potential scale changes in the future. How much room to scale have you built in? What is the first scalability wall you will run into? What happens when you reach that wall? How can you respond and remove the barrier without requiring a major rearchitecture of the application?

By thinking about how your application will grow long before it grows to those painful levels, you can preempt many problems and build and improve your applications so that they can handle these growing pains safely and securely.

CHAPTER 12

Getting Started Architecting for Scale with the Cloud

Awareness and knowledge of the cloud has grown significantly in the last several years. It wasn't that long ago that "using the cloud" was something only progressive organizations considered doing or otherwise was limited to startups looking to reduce capital costs in infrastructure utilized.

But it did not take long for enterprises to realize the value of the cloud. Acceptance of the cloud by all but the most conservative enterprises in the last few years has made using the cloud mostly mainstream. Or at least the desire to adopt the cloud is now mostly mainstream.

For many enterprises, though, finding success in the cloud is still a daunting challenge. Too often, organizations set overly high expectations for the benefits of migrating to the cloud while underestimating the amount of work required in the migration itself and the impact the migration has on the culture of their company. An unfortunate result can be a vicious cycle of blame, finger pointing, and grasping for something—anything—that could be considered a victory.

When they find it, organizations may decide they've had enough and stop the migration process before they can take full advantage of the cloud. But by putting off real reform, they won't realize the cost and innovation benefits that drove the cloud migration project in the first place. As migration costs balloon and promised features, functionalities, and applications fail to materialize, companies can end up seeing the cloud as little more than an expensive boondoggle.

Why did this happen? Often, the biggest error comes from thinking about the cloud in the wrong way. Enterprise management tends to think of a cloud migration as a

simple "lift-and-shift" operation—simply move existing applications that are running in their own data centers directly to the cloud, with as few changes as possible.

However, real cloud success, at scale, requires much more than lift-and-shift. It demands successfully navigating the world of the dynamic cloud. The dynamic cloud doesn't just facilitate application scaling; it makes the process faster and easier. It also helps development teams respond to changes faster and to implement these changes more quickly. That's not a luxury—it's a necessity to ensure the availability of modern applications that exhibit extreme scaling needs and extremely spiky performance. When you don't know when your customers will use your application, it's hard to predict your static infrastructure needs. A dynamic infrastructure is required to meet the needs of these modern applications without wasting significant resources.

Using the dynamic cloud, however, takes a higher level of commitment to using your cloud resources effectively than does a simple lift-and-shift. That's because after a migration, the type of application and infrastructure visibility that is required changes. Many resources become dynamic, so keeping track of what resources are important for what purposes also becomes dynamic. Additionally, applications now run on an infrastructure outside of a team's direct control, a concept that is foreign to many large enterprises.

Fortunately, becoming proficient in the dynamic cloud does not have to be scary or dangerous. Adopting the cloud can be done safely and effectively, but it is a continual learning experience. Organizations must be willing to learn and adapt cloud offerings to match their needs and expectations with the reality of what the cloud can provide. There is a learning curve of cloud maturity that ranges from simple lift-and-shift to a full architecture rewrite and adoption of the cloud and all of its capabilities. This chapter discusses this cloud maturity curve.

Six Levels of Cloud Maturity

Critically, you can't expect to get there all at once. There are six basic maturity levels that organizations go through during their cloud adoption process:

- Level 1—Experimenting: What is the cloud?
- Level 2—Securing the cloud: Can we trust the cloud?
- Level 3—Enabling servers and SaaS: Lift-and-shift, confirmation the cloud works pretty well
- Level 4—Enabling value-added services: Dynamic cloud becomes a practice
- Level 5—Enabling unique services: Dynamic cloud is deeply ingrained in the culture
- Level 6—Mandating cloud usage: Why do we need our own data centers?

To be successful in moving to the cloud, organizations must realize that this continuum of cloud maturity exists and understand the implications for their actions and processes. Moving from one level of maturity to the next isn't always easy, it isn't always fast, and the specific details differ for every organization. Also, organizations sometimes settle on a level of cloud maturity that's right for their culture but short of the end goal. That's OK, if that meets the expectations and requirements of their business.

Level 1: Experimenting with the Cloud

This first tentative step into the cloud relies on safe technologies—technologies that apply in simple ways to applications and parts of applications that are typically less mission critical.

Level 1 involves using the cloud for a single, simple piece of an application to test how cloud services work. Often, the first service used is a storage solution such as Amazon Simple Storage Service (S3), because it's easy to store some things in the cloud and avoid addressing the complex processes and systems needed for cloud-based computation, such as cloud-based servers and serverless computation.

This level usually starts as a one-off experiment, where one or more teams conduct stand-alone migrations. No cloud policies are created at this point; instead it's all about figuring out exactly what the cloud is.

Level 2: Securing the Cloud

This is a critical evolution point in an organization's cloud culture, as it begins to involve disciplines throughout the company—legal, finance, security, and so on. Trust becomes a core question at this point. Can we trust depending on the cloud for our business success? Can we trust putting our data in the cloud? Do we know how and where it is appropriate to trust and how to ensure that the cloud is secure enough to meet our needs?

This is when policies on how the cloud can be used within a company begin to be formed. The precise nature of these guidelines, from formal policies to ad-hoc "company culture" understandings, doesn't matter that much. What's important is that the entire company is involved and all stakeholders have input.

Level 3: Using Servers and Applications in the Cloud

The third stage of cloud maturity comes when an organization begins to replace on-premise servers and other backend resources. These are still simple lift-and-shift applications, with a basic philosophy of "Let's just move an application to the cloud and see what happens."

At this level, the goal is to understand how the cloud works for an entire application. This is where the organization begins to enjoy actual advantages from using the cloud, such as reduced cost and increased flexibility.

Enterprises need to be careful here, however. Level 3 can be a danger point. If enterprises attempt to determine the value of using the cloud to run their applications at this stage, they may find they're bearing the costs of the cloud without enjoying the corresponding benefits. That can cause companies to give up and regard their entire cloud effort as a failure. The solution is to use this level not as an end point but as a transition point. Avoid the temptation to stop once you've completed a lift-and-shift and say, "That's enough—we're now in the cloud." It's important to go the next steps and take advantage of the capabilities the cloud provides.

Level 4: Enabling Value-Added Managed Services

Here is where some of the inherent value of the cloud begins to appear. At this level, organizations start to look at cloud managed services, such as managed databases. Managed database services such as Amazon Relational Database Service (RDS), Amazon Aurora, and Microsoft Azure SQL provide database capabilities to applications while requiring less overall management. Rather, you let the cloud provider manage the database. Organizations may also look at services such as Amazon Elasticsearch, Amazon Elastic Beanstalk, and Amazon Elastic Container Service (ECS) to provide managed computation.

As the dynamic cloud starts taking effect here, the cloud's biggest benefits kick in. This is also when companies commit to using the cloud for at least some of their strategic applications and services.

Level 5: Enabling Cloud-Unique Services

Once a company becomes a cloud-enabled organization, it can look to leverage high-value, cloud-specific services. Uniquely available in the cloud, these services are designed specifically for the dynamic cloud. Some examples of services utilized at this level include serverless computing such as AWS Lambda or Microsoft Azure Functions, highly scalable databases such as Amazon DynamoDB, and other generalized services, such as Amazon Simple Queue Service (SQS) and Amazon Simple Notification Service (SNS).

At Level 5, the concept of dynamic cloud becomes embedded in an organization's application development and management processes. Use of these services also begins to tie the enterprise to specific cloud providers. While many cloud providers offer serverless capabilities, each one does so in a slightly different manner. As organizations begin to use these higher-value, cloud-unique services, they become tied not only to the cloud but also to specific cloud providers.

Level 6: Cloud All In

This is the ultimate maturity level of cloud adoption. At this topmost level, organizations begin to require use of the cloud for all new applications and begin to require existing applications to be migrated to the cloud. The usual end goal for customers at this level is to get rid entirely of their own corporate data centers and depend on the cloud for all infrastructure needs.

This level is especially common in cloud-native companies—the needs of legacy applications complicate the ability to be all in on the cloud. It's significantly easier for cloud-native companies to mandate all-cloud-based applications. More established enterprises may choose to retain their legacy data centers. However, more and more traditional enterprises are taking the cloud plunge and abandoning the business of managing their own data centers.

Organization Versus Application Maturity Level

The six cloud computing maturity levels apply to individual applications, organizations, or entire enterprises. But the cloud maturity level of a particular application may be higher or lower than that of the organization as a whole.

For example, early candidates for cloud migration include internal applications, because they present less risk of negatively impacting customers and the business itself. In fact, an internal application may jump to Level 6.

Larger, more complex legacy enterprise applications may be significantly slower in their cloud adoption strategy.

Meanwhile, the overall enterprise itself might still sit at Level 1, 2, or 3 and never make it to Level 6.

This is entirely normal and expected and demonstrates the complexities of cloud adoption in large enterprise organizations.

Cloud Adoption Mistakes

When you adopt the cloud, it's really easy to fall into a few very specific traps that can lead to significant problems in your adoption strategy. These mistakes are often the cause of a failed migration, or at least a perception that the migration was unsuccessful. They can also cause the cost-to-benefit ratio to skew away from the true value of the cloud. Be careful not to fall into any of the following traps as you look at performing your cloud migration.

Trap #1: Not Trusting Cloud Security

One of the biggest misconceptions that companies new to the cloud deal with is the issue of trusting the cloud. This shows up in many different ways, but dealing with security is a main one.

Security is very important to nearly all companies. Moving to the public cloud means taking an application that is safely behind the company's firewall and putting it on a public cloud. The first time you consider doing this, it'll seem scary. Can you trust the cloud to keep your data secure? Is your application safe from attack in such a public environment?

The short answer? Yes.

For the vast majority of companies, your company is probably safer in the hands of a public cloud provider than it is behind your own firewall. Why is that true? Because cloud service providers make a living on trust. They would not be in business if they could not keep their customers' data secure.

Cloud providers invest heavily in building high-quality security teams that spend their time advancing the state of the art in security protocols and procedures. By putting your data in the hands of a reputable public cloud provider, you take advantage of the learnings and best practices created by the leaders in the security field. Unless your company has the same resources to invest in security as the cloud providers do, your company can benefit from these learnings in so many ways.

By using a public cloud provider and taking advantage of all the security offerings it provides, you can actually keep your applications and data safer in the public cloud than you can behind your own firewall.

Trap #2: Performing Cloud Migration via Lift-and-Shift

Early in the process of adopting the cloud, many companies consider moving applications to the cloud by simply taking the application off of servers in their own data center and moving them to servers they've created in the cloud.

This type of migration is called lift-and-shift, and we discussed it earlier in this chapter as a cornerstone of one of the maturity levels of cloud adoption.

While lift-and-shift is a valid way to very quickly get your application out of your data center and into a cloud-based data center, it doesn't do anything to make your application cloud friendly. It doesn't do anything to take advantage of the native value and native characteristics of the cloud. Yes, there are some benefits you can get from a lift-and-shift migration, including the ability to expand to additional data centers simply by launching servers in another region. But that is about where the benefits stop. In fact, the cloud can actually be worse at this type of basic application hosting than your own data centers. Why? Cost.

The cloud can and does provide significant cost benefits for users that take advantage of the dynamic allocation capabilities of cloud resources. But it typically can't compare in cost to the basic, static infrastructure provided by a noncloud data center. When you use the dynamic capabilities of the cloud, you can save money. If you simply lift-and-shift, you typically don't save money and often spend more.

Doing a lift-and-shift can cost you money and time and not give you any of the benefits you were wanting with a cloud migration.

Trap #3: The Lure of Serverless—Depending Too Much on the Hype

It's easy to get caught up in the hype of the cloud, and the latest and greatest cloud service offerings often seem like the solution to all your problems. However, like with any new technology, understanding how and where to apply the technology is critical to successfully using it. This most certainly applies to the newest Function as a Service (FaaS) offerings by cloud providers, such as AWS Lambda and Microsoft Azure Functions.

These offerings promise the ability to provide an execution environment for your software without the need for managing the servers they run on. This "serverless computing" offering is very attractive to companies that are wanting to use the cloud to reduce their infrastructure management costs. But, like all new technologies, FaaS offerings such as Lambda are good for some classes of problems and not good for other classes of problems.

Yet I often hear statements from individuals such as "Lambda will solve my computing infrastructure problems" and "We're moving all of our software to Lambda."

To people thinking that FaaS offerings such as Lambda are a solution to all your problems, I say be careful. AWS Lambda and the equivalent offerings by other cloud providers give a huge advantage to a certain class of computing environments, but they can be overused.

If they are force fit into solving problems they weren't designed to solve, they actually can create more problems for you and your infrastructure management than they solve.

Use them as an important part of your application architecture, but don't depend on them to solve all your computing problems. Use them only where they make sense.

When and How to Use Multiple Clouds

When deciding to move an application to the cloud, you need to consider many factors before choosing a cloud provider. What features do you need? Which cloud is faster? Which one is cheaper? Which one is more reliable?

But here's another question that is being asked more and more often: how many cloud providers should I use?

The seemingly obvious answer is a single provider, but cloud customers cite a number of reasons to use multiple providers. First, some features you might want to use might be available on only one cloud provider, and other features might be only on another cloud provider. Another reason is that utilizing multiple providers instead of being tied to a single provider might offer better negotiation room when dealing with contracts. Yet another reason often cited is reliability—when one cloud provider goes down, the other cloud provider will still be available. Or the reasoning may just be seemingly random…part of your organization prefers one provider and part prefers another provider.

But your answer may or may not be the right one for your organization. Using multiple cloud providers may give you benefits, or it may actually hurt you, when you are doing it for any of the reasons just mentioned.

Let's take a look at what goes into making the best decision for your particular situation.

Defining What We Mean by Multiple Clouds

Before we talk about how many clouds you need, we need some definitions. The actual set of advantages and disadvantages of a multi-cloud arrangement depends greatly on the type of multi-cloud environment you are considering. So let's look at three different types of cloud configurations: joint cloud applications, selective cloud applications, and single cloud applications.

Joint cloud applications

A joint cloud application is when a single application uses two or more cloud providers to provide parallel capabilities. A given application or its services can run on any or all of the supported cloud providers, as shown in Figure 12-1.

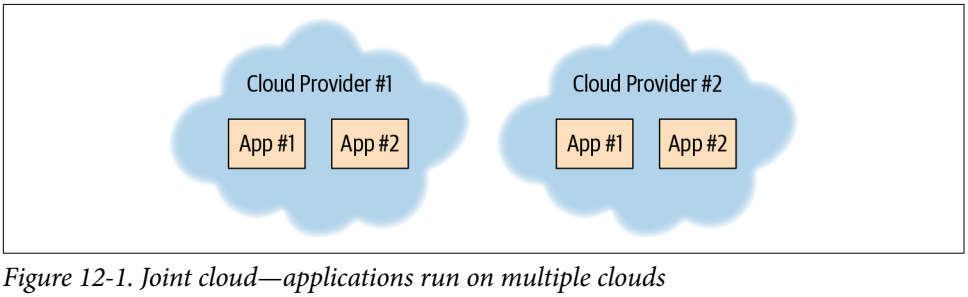

Figure 12-1. Joint cloud—applications run on multiple clouds

App #1 and App #2 can be run on either of the cloud providers. You can also load balance each application across both clouds simultaneously, if desired. If one cloud

becomes unavailable, the other cloud can take over responsibility for running the application.

Each application must be designed to run on either cloud provider, and the application can use either available provider to satisfy a given request. If one provider is unavailable, the other provider can take over to process requests for the application.

One major advantage of this approach is application resiliency. If a cloud provider experiences an issue, the application workload can be rerouted to the other cloud provider easily and relatively quickly. This lets the application continue functioning even if one provider has a massive failure.

This architecture is often cited as a solution to single-vendor cloud lock-in because you can easily switch your load between multiple cloud providers. However, this architecture also has significant disadvantages. For example, each development and operations team supporting the application must have an understanding of the workings of multiple cloud providers. This knowledge and understanding does not come for free. Similarly, each application must be tested and maintained on multiple cloud providers.

Additionally, when applications are written to support multiple cloud providers, they cannot take advantage of deeper feature capabilities provided by one particular provider. The application must be written to use the least common capabilities inherent in all the cloud providers being utilized.

In most cases, the shortcomings of this approach outweigh the perceived improvements in resiliency.

Selective cloud applications

This is when your company maintains relationships with multiple cloud providers, but any given application runs entirely on a single provider, as shown in Figure 12-2.

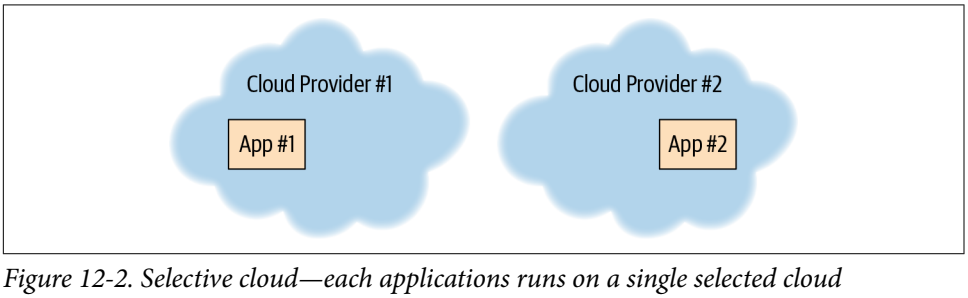

Figure 12-2. Selective cloud—each applications runs on a single selected cloud

You can see that each application is hosted on only a single cloud provider, but different applications may be hosted on different cloud providers.

In this scenario, a given component is designed to run and capable of running on only a single provider's cloud.

One perceived disadvantage of this approach is resiliency. If a cloud provider becomes unavailable, then the applications or services running on that provider will stop functioning. You cannot simply move traffic over to an alternate cloud provider. This is typically more of an intellectual issue than a practical one. It is rare for an entire cloud provider to become unavailable. Typically, only one or more availability zones or regions become unavailable. A properly written application can take advantage of multiple availability zones and regions to improve application resiliency without having to resort to using multiple cloud providers.

A real potential advantage of this architecture, though, is that individual applications or services can independently select which cloud provider they want to use based on whatever criteria makes sense for the application or service's owning team.

This architecture requires that each individual development and operations team supporting a given application learn and understand only how the single cloud provider it works with operates. Additionally, each team can take advantage of deeper feature capabilities unique to its specific cloud provider. The applications themselves can be designed, built, and optimized using cloud provider–specific best practices.

However, in this architecture the company must maintain multiple vendor relationships and agreements with each supported cloud provider. This is more of an administrative problem than a technical one, but it might be an issue for your organization.

In most cases, this approach gives you the desired flexibility of multiple clouds and the ability to do deep integrations with your cloud providers, without the application-level complexity that joint cloud applications require and without significantly sacrificing application resiliency.

Often organizations back into this particular architecture. One team or organization selects one cloud provider for its applications, while another team or organization selects another cloud provider for its own applications. The enterprise as a whole is multi-cloud, but individual applications are each single cloud.

Single cloud applications

This is the simplest design, in which a single cloud provider is used for all cloud needs within the company, as shown in Figure 12-3.

In this architecture, the company standardizes on a single cloud provider. It allows all development and operations teams to focus on the capabilities of that one provider. Knowledge can be easily shared across teams, and multiple teams can leverage a single set of best practices. And all applications can take advantage of the provider's full set of features.

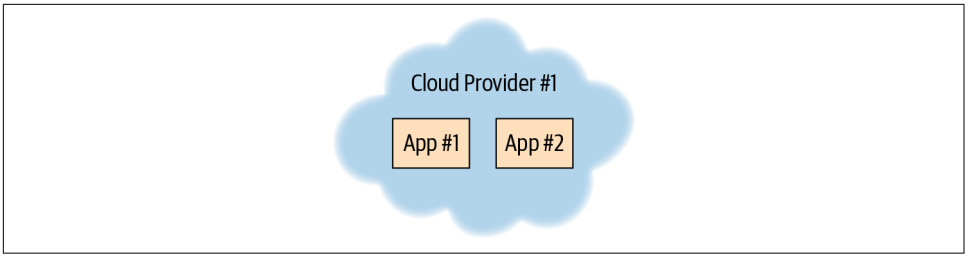

Figure 12-3. Single cloud—applications all run on a single cloud provider

From a management perspective, having a single cloud provider simplifies vendor management. Additionally, since all traffic goes through a single provider, that provider has a higher volume usage, which may allow you to negotiate better pricing and other terms.

However, this solution obviously requires a strong commitment to a single cloud provider, which can be problematic for some companies. When a problem does occur, it is much harder to negotiate a solution when you are locked into a single vendor.

This solution may be the simplest to manage and control of all the options, but it limits the flexibility of your development teams.

Which Model? Which Cloud?

So which cloud model should you use? What makes sense for your company? The final answer depends on the needs of your company and your applications.

From an application perspective, the advantages of having an application runnable on multiple cloud providers is typically outweighed by the cost and complexity associated with maintaining multi-cloud-capable applications. Therefore, in almost all cases, the joint cloud applications model shown in Figure 3-1 does not make sense. If you are worried about maintaining high availability within your applications while tying them to a single vendor, consider using the high-availability solutions available from that vendor. For example, simply using multiple availability zones and multiple regions for your application running on AWS can dramatically improve your application's resiliency without incurring the costs and reduced capabilities of making your application run on multiple independent cloud providers.

When selecting a specific cloud provider, you should look at the following:

History of reliability
: How reliable has the service been historically? How quickly are outages dealt with? Do outages impact the entire provider, or do they impact only specific regions at any one time (allowing you to use multiple regions to improve resiliency)?

Capabilities of availability technologies
: Does the provider give you multiple availability zones and multiple independent geographic regions to allow fault isolation and failover? How independent are the zones and regions?

Availability of services
: Does the cloud provider have the types and depth of services you require?

Reason for moving to the cloud
: Why are you looking to move to the cloud? Is it to accelerate innovation or to help you in scaling? Whatever the reason, make sure it matches the cloud provider's capabilities.

If you choose to use multiple providers, do so because it makes sense for the given capabilities you require from those cloud providers. Don't do it to increase resiliency by using multiple providers. The reality is that the benefits are outweighed by the costs and disadvantages.

The Cloud in Summary

This chapter focused on how to use the cloud to architect scalable applications. We discussed how organizations mature in their ability to use and trust the cloud; common mistakes that organizations make when they adopt the cloud and the traps these mistakes can lead them into; and how and when to use multiple clouds in your applications.

About the Author

Lee Atchison is the senior director of cloud architecture at New Relic. For the last eight years he has helped design and build a solid service-based product architecture that scaled from startup to high-traffic public enterprise.

Lee has 33 years of industry experience, including seven years as a senior manager at Amazon. At Amazon, he led the creation of the company's first software download store, created AWS Elastic Beanstalk, and managed the migration of Amazon's retail platform from a monolith to a service-based architecture.

Lee has consulted with leading organizations on how to modernize their application architectures and transform their organizations at scale, including optimizing for cloud platforms and service-based architectures, implementing DevOps practices, and designing for high availability.

Lee is an industry expert and is widely published and often quoted in publications such as *InfoWorld*, *ComputerWorld*, *Diginomica*, *IT Brief*, *ProgrammableWeb*, *The New Stack*, *CIOReview*, *DevOps Digest*, and *DZone*. He has been a featured speaker at events across the globe from London to Sydney, Tokyo to Paris, and all over North America.

Colophon

The animal on the cover of *Architecting for Scale* is a textile cone snail (*Conus textile*). It is also known as the "cloth of gold cone" due to the unique yellow-brown and white color pattern of its shell, which usually grows to about three to four inches in length. The textile cone is found in the shallow waters of the Red Sea, off the coasts of Australia and West Africa, and in the tropical regions of the Indian and Pacific Oceans.

Like other members of the genus *Conus*, the textile cone is predatory and feeds on other snails, killing its prey by injecting them with venom from a radula, an appendage that resembles a small needle. The conotoxin used by the textile cone is extremely dangerous and can cause paralysis or death.

Their shells are sometimes sold as trinkets, but textile cones are plentiful, and their population is not threatened or endangered. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Wood's Illustrated Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.