O'REILLY®

Overcoming IT Complexity

SIMPLIFY OPERATIONS, ENABLE INNOVATION, AND CULTIVATE SUCCESSFUL CLOUD OUTCOMES

Early Release
Raw & Unedited
Compliments of

f5®

LEE ATCHISON

with contributions from MARK MENGER

A big yes to modernization. But what about complexity?

As applications become more distributed than ever, complexity continues to rise. And with 84% of organizations planning moves toward the edge, this complexity will only increase. Read these and other trends in the latest F5 State of Application Strategy Report.

Get the report ›

Overcoming IT Complexity

Simplify Operations, Enable Innovation, and Cultivate Successful Cloud Outcomes

Lee Atchison
with contributions from Mark Menger

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

Overcoming IT Complexity

by Lee Atchison

Printed in the United States of America.

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

Editors: Jennifer Pollock and Gary O'Brien
Production Editor: Beth Kelly
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: Monica Kamsvaag
Cover Designer: FILL IN COVER DESIGNER
Illustrator: Kate Dullea

March 2023: First Edition

Revision History for the Early Release

2022-07-07: First Release
2022-08-08: Second Release
2022-09-21: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492098492* for release details.

978-1-492-09843-0

[FILL IN]

Contents

What is the Modern IT Complexity Dilemma?

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

The complexity of modern IT systems supporting modern applications can impact your customers, your partners, your employees, and the quality and security of your application.

Addressing this complexity dilemma is essential, but it is not easy. The IT Complexity Dilemma refers to the challenges businesses face when trying to manage and optimize their IT operations. The increasing complexity of modern IT systems has made it difficult for businesses to keep up with technology changes and make necessary updates to their infrastructure, making it difficult for them to realize their desired return on their technology investments. Business pressures have caused IT organizations to focus increasingly on creating new features and focus less on resolving ongoing problems and upgrading existing architectures. This has led to a build-up of technical debt, which can have consequences such as slowed business processes, increased IT costs, and even system failures.

There are several measures that businesses can take to mitigate the effects of the IT Complexity Dilemma, including investing in automation technologies, establishing standard operating procedures, and hiring skilled IT professionals. By taking these steps, businesses can improve their ability to manage and optimize their IT operations, minimizing the negative impacts of complexity.

However, even with these measures in place, businesses will still face challenges in managing their IT operations due to the increasing complexity of modern IT systems. By attempting to mitigate the effects of this complexity, businesses can improve their efficiency, security, and competitiveness in today's increasingly complex IT environment.

As we will discuss later in this chapter, the core to the complexity dilemma is this technical debt. In fact, technical debt and complexity go hand-in-hand. We'll learn that, beneath the surface, technical debt is much more than needed code refactorings.

Technical debt and complexity go hand-in-hand.

Instead, technical debt is a metric that describes everything that interferes with the smooth operation and customer experience of an application. In other words, technical debt is everything that makes the application complex—whether that is operational complexity, complex customer experiences, or simply complexity that makes the application difficult to enhance, expand, and improve.

But before we can talk about complexity, it's important to understand how IT organizations are structured in modern enterprises, and how the operations and development teams within the IT organization interact to create a working system.

Structure of Modern IT Organizations

IT organizations vary considerably in size and shape. A modern IT organization is typically a relatively flat, matrix-type structure. Teams of developers and operators work together closely in this organization. To keep up with the fast pace of customer demand and technological change, both the development and operations teams need to adapt quickly. Development needs to both quickly build new systems and applications and be able to repair and upgrade existing systems. Operations not only needs to deploy and manage those systems quickly but also detect and resolve problems when they occur. A close, working relationship between development and operations is critical to this speed.

However, natural separations start taking shape as applications grow in complexity and the organization grows in size. Traditional divisions between development and operations begin to formalize, and the space between the two groups grows and expands.

A flat management structure has been able in the past to assist the organization in keeping the communications channels flowing as much as possible—flowing between development, operations, security, and product leadership teams. But, as the organization grows, keeping the organization flat and responsive becomes harder and harder.

Management and organizational structures are required to keep the growing organization operational. Formalized processes help yield consistent results and plans. Yet, these same structures and processes create a natural blockage to communications flow. This blockage makes

it harder for the organization to function. Teams split, and the organizational distancing limits cooperation and communications. This limits growth.

Ironically, the biggest inhibiter to growth is, in fact, growth.

Ironically, the biggest inhibiter to growth is, in fact, growth.

The organizational structure gets more complex, and the application gets more complex.

Since an IT organization is only as good as the management that drives it, it is essential to have a strong and effective management team in place. This team is responsible for steering the organization in the right direction, setting goals and objectives, and ensuring that all aspects are running smoothly.

The management team must also adapt to changes quickly and effectively respond to new demands in the marketplace. They must work across organizational boundaries, and operate in unison with all product, development, operational, and support teams.

How the IT organization is structured varies considerably based on the nature of the business. The type of company is the biggest indicator of the structure of the IT organization. And nothing drives the organization more significantly than where and how the software development teams are organized.

ROLE OF SOFTWARE DEVELOPMENT IN IT ORGANIZATIONS

Within an IT organization, the structure and responsibility of the development organization is related to the nature of the business and the structure of the rest of the company. The importance of the development organization ranges from a limited role in managing internal processes and systems to a role of being an integral part of the core business model of the company. This means there is no one-size-fits-all organizational structure that defines a modern IT organization.

But we can generalize. For this discussion, we're going to define three types of businesses and describe the IT organization structure that tends to occur in each of these businesses:

- The SaaS Focused IT Organizations
- The Non-SaaS Software Focused IT Organizations
- The Non-software Focused IT Organizations

Let's look at each of the three types individually and their characteristics. Then we will look at how the IT and software development organizations look different inside each type of company. In the end, you will find that your circumstances lead to an amalgam of two or three of these types.

Business Type #1: The SaaS Focused IT Organizations

This is a business where the primary purpose is to create a Software as a Service (SaaS) application, or a company where the operation of customer software is the primary goal of the business. This includes business-to-business (B2B) companies providing services such as inventory management, financial planning, communications, and sales infrastructure. Examples of SaaS companies and products include Intuit QuickBooks, Slack, Shopify, MailChimp, and Salesforce.

There are also business-to-consumer (B2C) SaaS companies providing entertainment, retail shopping, and social media services. Example B2C SaaS companies include Amazon, Netflix, Facebook, and Twitter.

A highly functional SaaS organization requires a high caliber application development team, **and** a high caliber operations organization. The two teams must cooperate to succeed. DevOps is a common model in a modern organizational structure.

Take a look at the example SaaS company in Figure 1-1. The engineering and product management of a SaaS company are typically proportionately large groups that require large investments in application development. Each team owns a part of the overall SaaS application and builds, tests, deploys, operates, and maintains only the services they are responsible for. Operations teams provide a smooth operations infrastructure that supports the development teams. The focus of the enterprise, typically, is on the software development teams. These companies may have separate IT organizations to support business processes, but the application development teams are not part of that organization.

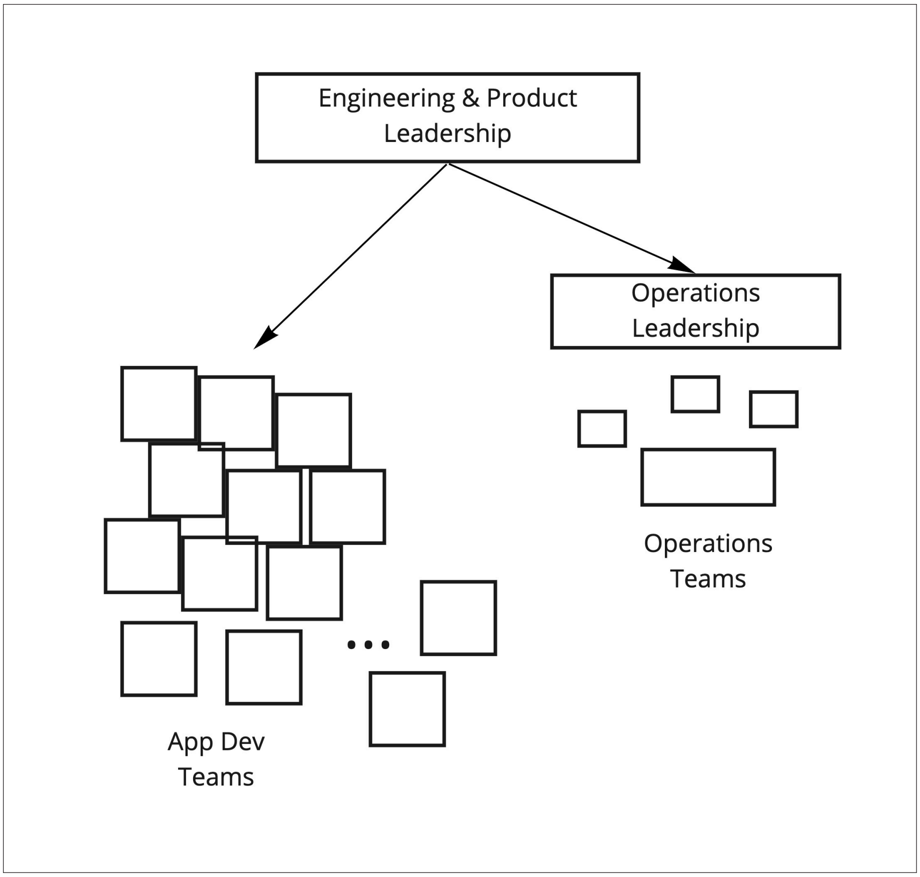

Figure 1-1. SaaS Focused Organization

In these high tech, software driven companies, the application development teams are a core component of the enterprise. Given its importance, this group reports high in the company's organizational hierarchy.

Business Type #2: The Non-SaaS Software IT Organizations

This is a business with the primary purpose of creating software that they sell to other people or companies, but they typically do not operate the software for their customers. Any software that doesn't require a significant back-end SaaS-like application to function requires the software they produce to be operated directly by the customer.

Examples of this type of software include popular software such as Microsoft Word and Adobe Illustrator. But it can also include game software such as Angry Birds, music applications, and ebook readers. It might include tools like anti-virus software, firewalls, and news aggregators.

A non-SaaS software organization has a software or engineering focused mission. These organizations have high caliber applications development teams, but they typically do not have a strong operations team. Since the software the organization sells is given to customers to run, rather than operated by the company itself, there is no need for a large operational focus.

A generic separate IT Organization provides tools and processes that support the company as a whole, including the product development teams and sales/marketing, etc. However, these teams are support teams. They support the efforts of the product development teams, but they do not contribute to the product development or operations at all.

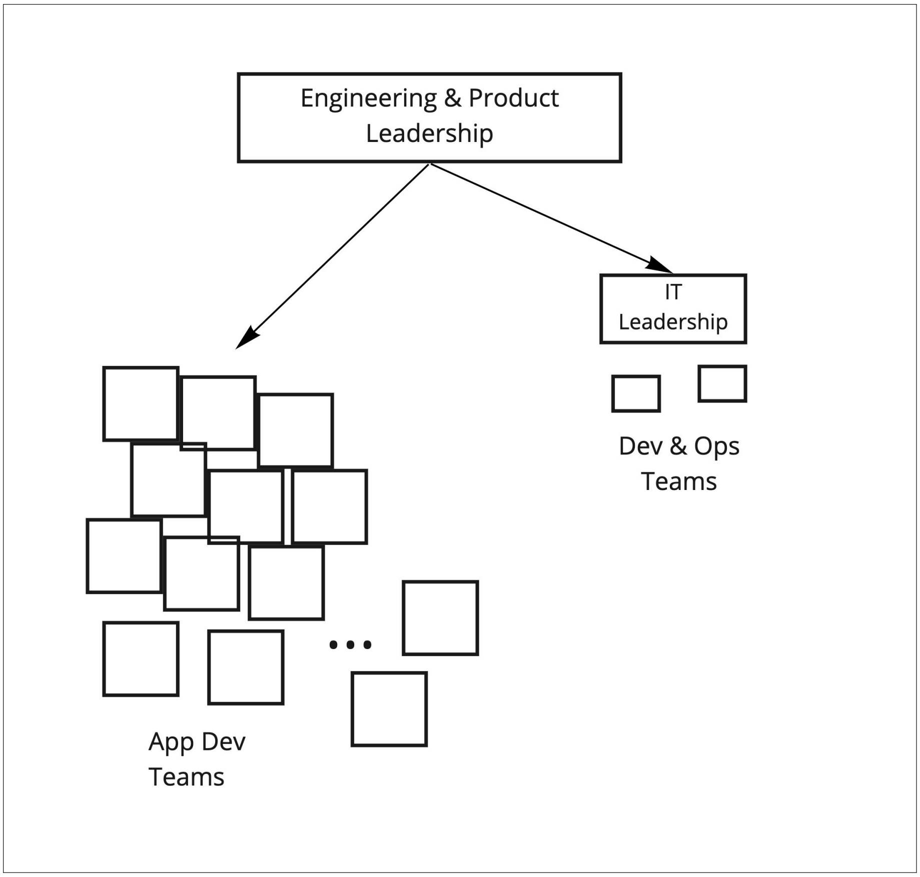

Figure 1-2. Non-SaaS Software Company

Take a look at the example in Figure 1-2. Non-SaaS software companies have the same focus on application development teams as SaaS focused organizations, but these teams are typically not DevOps teams. They are independent software development teams that produce software sold and delivered to customers. The IT organization is small and provides support to the company as a whole, including development and operations of tools for the company. The IT organization is separate and isolated from the primary, mainstream product development software teams.

Business Type #3: The Non-Software Focused IT Organizations

This is a business with a primary purpose other than producing, selling, or operating software. They likely will use both internally and perhaps customer facing software to run their business, but their primary business is not software. This is almost all non-technology focused companies, including banks, restaurants, stores, taxi services, airlines, railroads, media companies, etc.

Notice that some companies do fit multiple categories. For instance, Microsoft offers SaaS services (Office 365) and non-SaaS software (Microsoft Word and Halo). Additionally, a company such as Charles Schwab may offer investment software as a service, yet they also focus on general financial services and investments. These companies may have different divisions that appear to be separate companies, each structured differently. Or they may have a hybrid structure. Keep in mind that these categories are generalizations.

The mission of a non-software company is not technology focused. They may use software as a tool internally to manage the sales, marketing, manufacturing, or other business processes, but software is secondary to their primary business.

As such, there are no large application development teams. There is an IT organization, and that organization will have a relatively small development and operations team within the IT organization itself. Calling the organization "small" is in relation to the company itself. Only a small portion of the company's resources are invested in IT systems and personnel.

Figure 1-3 illustrates this. The company leadership has bigger things to focus on, leaving IT leadership to manage these small development and operations teams.

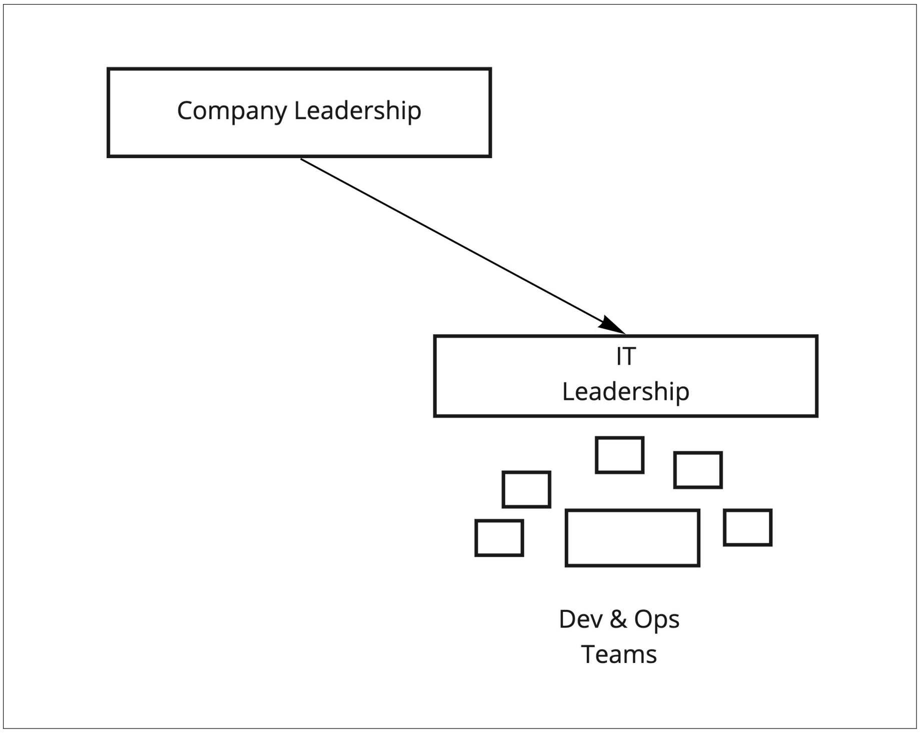

Figure 1-3. Non-Software Focused Company

For these more traditional non-software-based enterprise companies, the IT organization is purely in a supportive role. The IT organization is primarily operations focused, but may also have some development capabilities. Tooling and operational processes are the central focus.

THE STRUCTURE OF IT ORGANIZATIONS

Within an IT organization, the development teams are focused on driving the tooling and resources needed to operate the company, including building applications necessary for the company to run. The responsibilities and the processes the teams follow depend on where the company fits within the three types of organizations we just discussed.

Development – Company Focus and Talent Availability

There is a direct correlation between the amount of focus your company places on software development with the availability of high quality technical talent and software leadership available to your organization. The best software developers, architects, and software leaders tend to gravitate towards the much more lucrative opportunities in SaaS application development, and other software-centric companies.

This means that organizations where software plays only a secondary role in the company's mission find it difficult to attract and retain software talent. Often this means the organization, as a whole, suffers. Yes, there are high quality, talented developers in these organizations, but they are much harder to locate and hire, and hence tend to be less available to a typical non-development-centric organization. This tends to create less innovation and fewer creative solutions to problems in these types of organizations. Rather than state of the art software applications, the applications created in such organizations tend to be supportive applications that lack a high degree of innovation.

The caliber of your IT development teams is critically important in determining the sophistication of applications your organization can support, and your organization's ability to respond to the increase in complexity that occurs over time.

The result is, organizations where software is a secondary part of the business rather than a primary part tend to be organizations that are more sensitive to the IT complexity dilemma.

Operations — Different Focus than Development

Interestingly, a different characteristic occurs in the operations part of IT organizations. Operations organizations tend to attract high quality talent based on how central the operational aspects of the organization are to the business focus. This means SaaS companies, which are highly dependent on high quality operational organizations, tend to attract high quality operational talent, compared to organizations where the operations are secondary to the goals of the business.

The next highest talent attraction is to non-software companies that require a significant internal software infrastructure to keep the organization running smoothly. This includes organizations such as banks and other financial firms. These companies have important software infrastructures that must stay operational and attract strong operational-focused talent.

Less attractive to operational talent are companies that produce "boxed" software that is operated on customer computers and systems, rather than internal operational environments.

This makes sense. SaaS companies rely heavily on highly performant operational environments and invest heavily in these areas. This investment, and the opportunities it produces, attract the top operational engineering talent to these organizations. Organizations that are less operations focused need less investment in this area, and hence don't attract as much interest.

Traditional operations, however, is changing. Many traditional operational capabilities are handled by outsourced infrastructure, such as SaaS applications and cloud service providers. Additionally, newer tooling and capabilities automate a large portion of basic operational needs.

Tools such as *Infrastructure as Code* (IaC) and *Operations as Code* (OaC) help with this automation, and strive to make operational setup and basic operational responsibilities automated and repeatable. This improves overall operational reliability. Additionally, since Scripts and script-like descriptions drive iaC and OaC, these capabilities encourage code as documentation and knowledge sharing of the operational environments involved. Finally, since IaC and OaC generalize the operational aspects of an application into code-like capabilities, they allow the use of standardized and well understood development processes, such as revision management. Revision management allows tracking and correlating failures to changes, creating a better operating team, reducing mistakes, increasing security and traceability, and improving overall accountability.

The Role of DevOps in Modernization Enterprise

DevOps is a term in wide use in modern application organizations. It describes the collaboration between development and operations teams within an organization. The intent of DevOps is to break down the traditional barriers that exist between these two groups and encourage collaboration and cooperation, allowing them to work together more effectively and efficiently. This leads to faster problem solving, increased efficiency and responsiveness, and overall higher application quality and availability. DevOps is becoming the norm in modern IT organizations.

In a DevOps organization, individual teams own some portion of the application—both the development and testing of the component and the deployment and ongoing operation of the component. In modern applications, the components are typically application services, meaning the owning team is responsible for the development, testing, deployment, operation, and ongoing support of the services in their care.

Take a look at Figure 1-4. This shows a software company that uses DevOps principles. As such, it describes the same sort of SaaS company described back in Figure 1-1. The primary difference is that both the development aspects, and most of the operational aspects of ownership are assigned to the same team under the engineering and product leadership organization. There is still a very thin operations support organization, but the job of this organization is not to manage the operations of the application; rather, their job is more of a tools and infrastructure team. They provide tools and assistance to the product teams that own and operate their individual services.

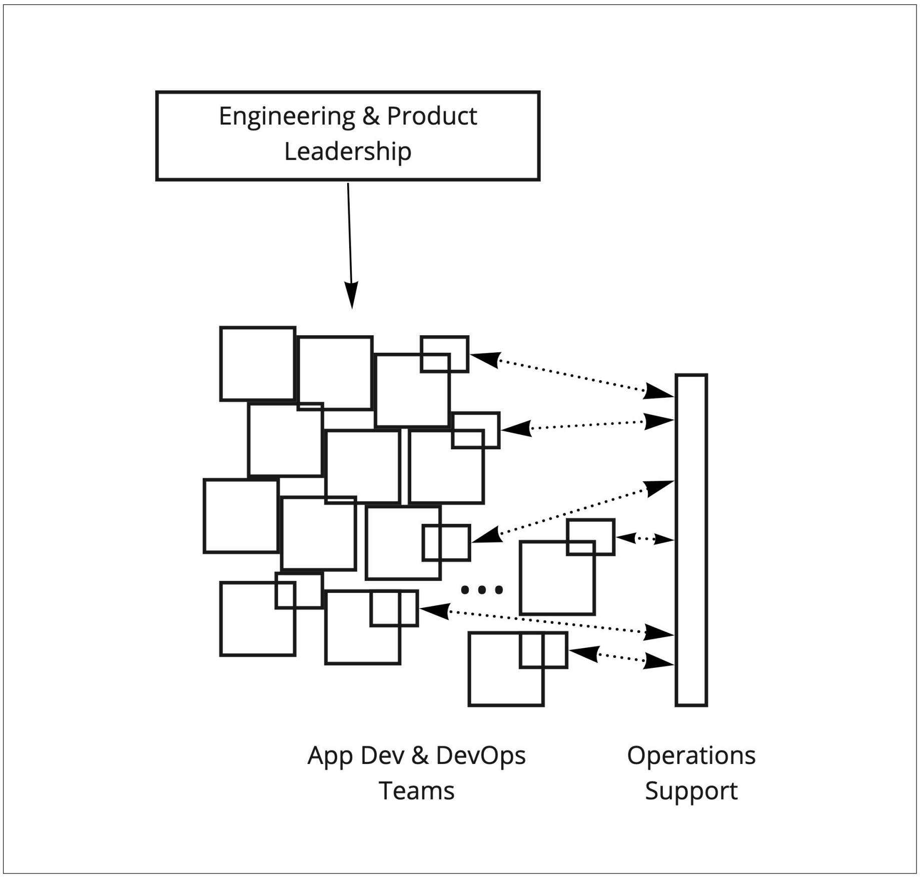

Figure 1-4. DevOps Focused Software Company

By merging the development and operational aspects of individual services into a single organization, communication barriers between the historically typical development org and operations org diminish, which improves the performance and reliability of the organization.

Now that we understand more about how modern IT organizations are structured, we can talk about the problem complexity plays in these organizations and how complexity impacts the long term viability and success of the organization.

THE ADVENT OF COMPLEXITY

It's hard to point exactly when complexity begins within a young startup IT organization, but it's usually tied to some decision that was meant to reduce time to market or cost to market, at the expense of some further work or cost later on. This starts the slow and inexorable climb in complexity. For larger, more established enterprises it's undoubtedly tied to the incorporation of technology into the established enterprise's processes. In either case, the increase in complexity is tied to the increase in technical debt. So the advent of IT complexity is driven by the collection of technical debt within the organization.

Technical Debt – The Key to Complexity

In software development, technical debt is the cost of additional rework caused by choosing an easy, limited, or sub-optimal solution now, rather than using a better approach that would take longer or cost more money.

Ward Cunningham coined the term technical debt. According to Ward:

"technical debt includes those internal things that you choose not to do now, but which impede future development if left undone."

The code development aspect of technical debt only captures a small part of it. Technical debt applies to all aspects of the modern application design, development, and long term operation of the application.

For SaaS and other cloud-centric applications, the long-term operational impact is a significant driver of technical debt. The longer an application operates, the greater the technical debt. The greater an application's technical debt, the greater the negative impact on the long-term operation of the service.

When a SaaS company decides to add a new capability to an application but takes shortcuts in the architecture to launch the capability quicker, they are adding to the technical debt of the application and the company. Unless a concerted effort is made to resolve the technical debt by completing the proper design and architecture, your application will build up this debt until it hinders your development and operational processes.

Technical debt is similar to financial debt. In moderation, it can be handled and the cost can be dealt with. But if technical debt is not repaid, it can accumulate "interest" in the form of additional technical debt. Just as in financial debt, too much interest and you can no longer afford to repay the debt. Too much technical debt makes the changes necessary to resolve (or pay back) the technical debt harder and more expensive to implement.

Let your financial debt grow too large, and you will go broke. Let your technical debt grow too large, and your application will become unsupportable and unsustainable.

How does technical debt grow? Technical debt can grow naturally and quietly during the normal product development process. Every project that contributes to a product, also contributes to its technical debt. This is illustrated with the top box in Figure 1-5. During the normal product development, work and output is added to the product, as well as some amount of debt to the stack of technical debt

Sometimes, a project is done “quick and dirty”, such as when a new feature is added without proper design in order to get it out the door quicker. In these cases, the project adds more debt. Sometimes, the project can even add more technical debt than useful capabilities. This is the example project shown in the middle box in Figure 1-5. More technical debt is added to your application than the amount of real value the project provided.

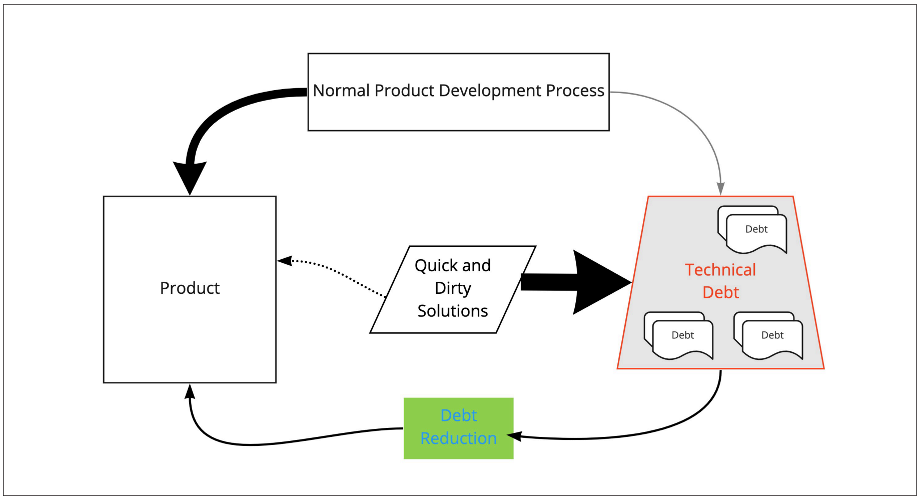

Figure 1-5. Flow of technical debt during product development

To keep technical debt from growing without bounds, some effort needs to be added to each project to reduce the technical debt. As shown in the bottom box in Figure 1-5, keeping your technical debt at a sustainable level requires constant investment in reducing the technical debt over time.

This constant flow of increasing and decreasing technical debt is one of the reasons why it can sneak up on a product. If more debt is regularly added than is reduced from the backlog, the debt will grow, yet the growth may not be noticeable. It's not until the debt has grown to a point where it starts having a negative impact on your product that you notice its size. At this point, it may be too large to deal with effectively and easily.

Each project can either increase or decrease the technical debt within a system. During a full, high quality project, the planned work often includes doing all the work necessary, along with working on reducing some amount of related technical debt. When the work is completed, the technical debt for the application is lower than it was before. This is illustrated in Figure 1-6. The **work completed** for the project is larger than the project itself, and the extra effort is towards reducing the size of the technical debt. This is a project that's dealing with technical debt in a healthy way.

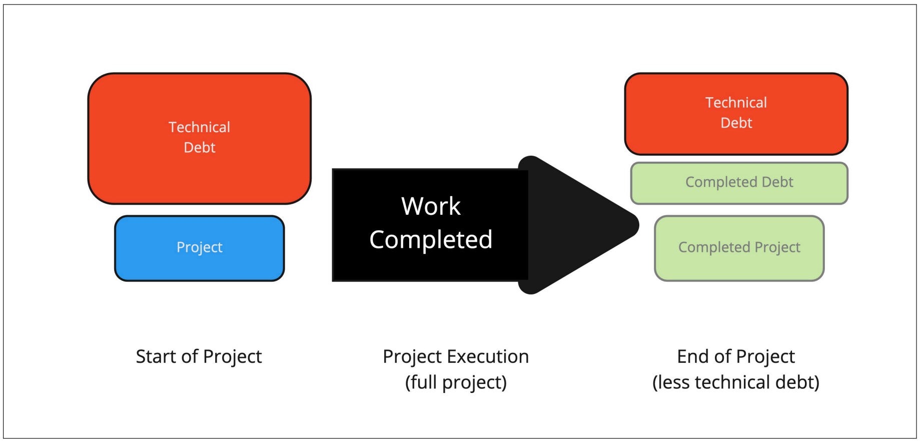

Figure 1-6. A full/complete project can plan to reduce technical debt

Unfortunately, many projects are much more quick and dirty. They are designed to only complete as much of the project as is absolutely necessary, leaving the rest of the project as work that will be completed later. In fact, a common project management philosophy involves building an MVP — Minimum Viable Product, essentially dictating that you do as little product work as possible to get a functional product out the door.

The result is work that is not completed. More often than not, this increases the overall technical debt of the application. This phenomenon is illustrated in Figure 1-7. Here, the **work completed** is only part of the total project. We have left some amount of **work not done** out of the project. This additional work which was not completed, ends up increasing the overall technical debt remaining in the project.

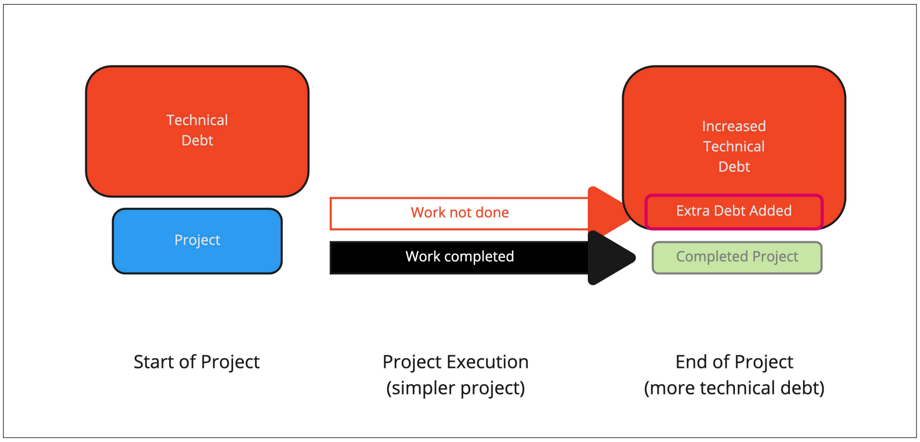

Figure 1-7. A quick & dirty project often increases technical debt

In any case, as projects are executed, the amount of technical debt can vary over time, sometimes decreasing, sometimes increasing. The more full, high quality projects that are completed, the lower the overall technical debt. The more quick and dirty projects used to implement functionality, the higher the resulting technical debt. The types of projects you execute will, over time, vary the total technical debt you have in your application.

The Negative Impact of Technical Debt Sometimes deciding to build a simpler solution now, in favor of delaying longer term implementation is advantageous (this is the Figure 1-7 situation). It allows you to get a solution out to customers earlier, which allows the company to start monetizing the change, and receive customer input on capabilities the customer likes and does not like, which can be fed into a later, more ambitious solution. This is analogous to saying that borrowing money is advantageous if you use the money to contribute to a greater cause, such as purchasing a home. Paying some interest on borrowed money is fine, as long as the money you borrowed is put to good use. So too, with technical debt, managing some technical debt is useful and appropriate as long as you give value to your product and your company. Technical debt becomes a problem when it is left unresolved—unresolved technical debt ages over time and increases in cost.

Using the financial metaphor, technical debt becomes a problem when it builds up so that the cost of servicing the debt is too great, and it impedes your ability to invest in future projects. So, too, technical debt becomes a problem when managing and servicing that debt is overwhelming compared to managing and servicing the product.

When too many quick and dirty projects rule your project plan, and projects designed to reduce debt are not staffed in your company, your debt starts to become unmanageable. If this goes on for too long, technical debt overwhelms the project, as shown in Figure 1-8.

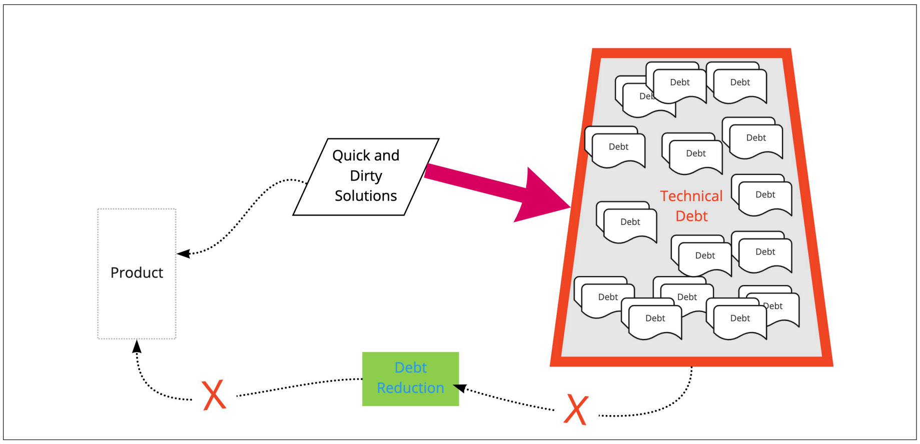

Figure 1-8. Technical debt overwhelming the project

In this scenario, servicing the debt becomes the dominant role of your team, and you spend little or no time contributing to improving the product. Your debt is too large to be effectively managed.

The organizational pain of complexity

Technical debt and complexity go hand-in-hand. So too do complexity and organizational pain. While technical debt is not a complete picture of application complexity, the growth of technical debt is tied very closely to the growth of application complexity.

As your application gets more and more complex, many things happen to it:

It becomes brittle

A complex application is subject to minor issues quickly escalating into major problems. While most applications operate in a well of a positive feedback cycle, a complex application's operational well turns into a negative feedback cycle, and minor issues quickly escalate out of control. Figure 1-9 shows a stable application tends to stay in the valley between the two hills, which builds success upon success, while the brittle application is ready to roll away off the top of the hill into failure at the smallest of nudges.

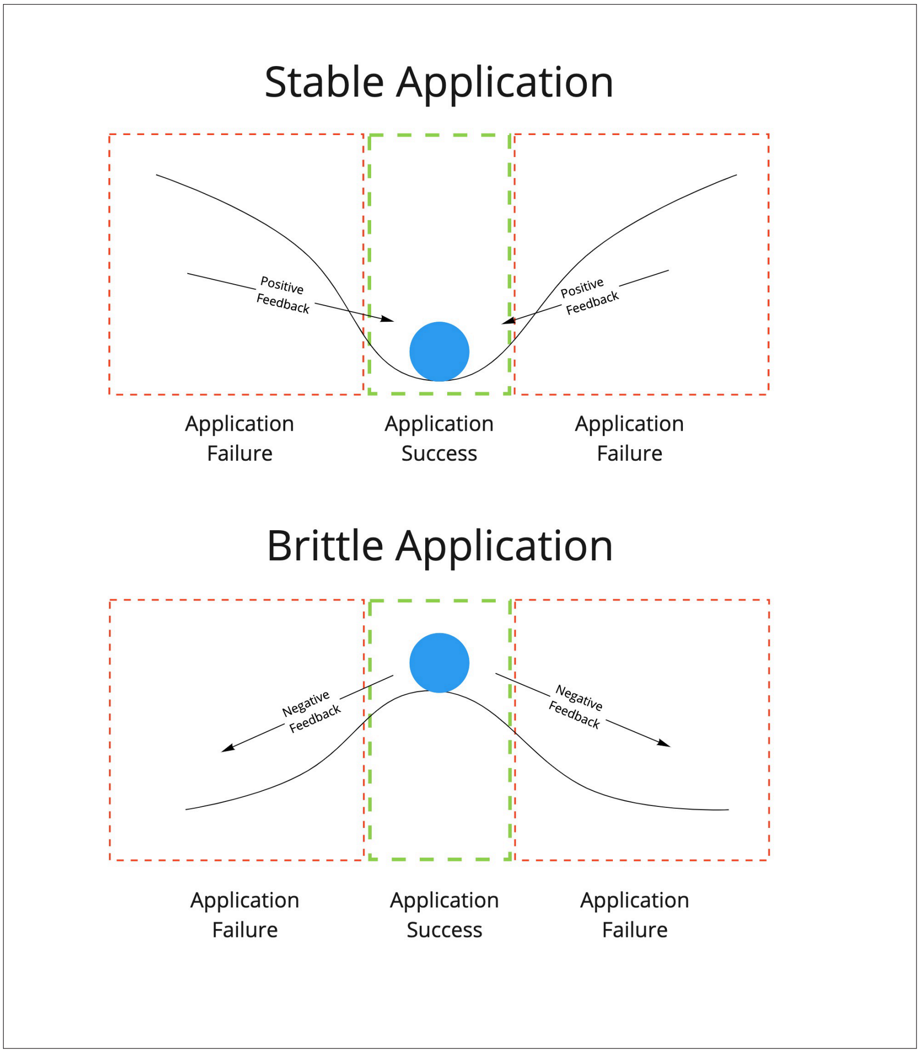

Figure 1-9. Application brittleness leads to instability and failure.

Fewer engineers have complete knowledge of the application
Only the most senior engineers, and those engineers working on the application the longest, have a broad understanding of how the complete application works. As time goes on, the knowledge of these most senior engineers becomes

diluted, less accurate, higher level, or more specialized. Broad, general purpose, but detailed knowledge of the application as a whole is no longer possible by single individuals.

The knowledge that engineers do have on the application becomes obsolete quicker

Complex applications change frequently, and engineers' knowledge about how the application works gets outdated quicker.

It gets harder to bring new engineers up to speed

Complex applications have long learning paths. This is not only because there is more to learn. The knowledge needed for new engineers to become productive is more distributed, anecdotal, and out of date.

The net result of these issues is higher organizational pain. This pain translates into poor quality changes, less motivated staff, and ultimate staff turnover. Higher turnover means greater need to train new engineers, which is harder as the pain increases. Brittleness leads to lower availability, and customer-visible issues and failures.

This is the pain of complexity.

Messy Desk Syndrome Imagine a perfectly clean desk. Now, take a sheet of paper and set it in the corner. Is your desk messy? No, not yet. Now you take fifty other sheets of paper that go together and sit them on top of the one sheet in the corner. Is your desk messy? No, not yet. Now imagine more sheets, but these don't go with the stack in the corner, they are for a different project. So you put them in different locations on the desk, just single sheets in single locations, seemingly in a location that makes sense. Now put more papers and documents and books and folders and pictures one at a time all over the desk. If you don't know where something goes, just put it in a new location. You'll figure it out later. Sooner or later, your desk is messy. In fact, it's extremely messy.

Unless you have a solid organizational plan for organizing the papers on your desk established *at the beginning*, and stick with it, sooner or later your desk will become messy, one sheet at a time.

Your desk becomes messy because you didn't have a plan from the beginning, but just "winged it" along the way. You made your desk messy simply by using and working on it.

So too, your organizational pain becomes large because you didn't have an architectural plan from the beginning. Rather, you started with no plan and adjusted and changed the plan as time went along. You "winged it", metaphorically. You have added technical debt, and hence organizational pain, simply by working and building the application.

Every action you take, little by little, builds up. Your technical debt grows a bit at a time until it becomes overwhelming.

- "Let's change our login process to allow saving login credentials in the user's browser"
- "Let's add this new feature to that menu"
- "Users would rather this feature work in three steps rather than the current four steps. Let's combine two of the steps."
- "We need to remove the per-session limit on this resource"
- "We don't have time to build this full feature now, but we can build this smaller feature, which will make many customers happier. We can do the rest later."
- "Let's release this feature this way first, and then we can collect input from customers and modify it to make it more user friendly as we get more input"

Any one of these statements can correspond to a simple set of changes that makes perfect sense at the time. It might not have any obvious impact on overall technical debt at all.

But the little changes...and the little debt...and the little impact...and the little piece of paper on the corner of your desk...adds up. And like the messy desk, each action may individually seem perfectly benign. Actions may look perfectly acceptable. But, when combined, they multiply and become overwhelming.

Icing the Cake

Many IT organizations use the expression "icing the cake". Icing the cake is when you describe the current situation as "everything is ok", whether it actually is ok or not.

The slow accrual of technical debt and organizational pain leads to this process of icing the cake. At the start, all is well and it's easy to say "everything is ok", because it is—or at least appears to be.

But as time goes on, and technical debt increases, and organizational pain increases, little by little, things aren't ok any more. But the tendency to keep saying "everything is ok" is strong. The organization keeps "icing the cake".

Ultimately, a debt-ridden application operated by a pain-ridden organization still says that "everything is ok". They fail to notice what's obvious to every outsider. The pain is real and the organization is not ok. Without you noticing it, the icing went bad.

Complexity in an IT Organization

Complexity grows in IT organizations as well. Complexity starts by growing within your application. As your application grows complex, so does the infrastructure needed to run the application. Your IT operations become more complex. Your engineering organization becomes more complex. To wrap their minds around all of this, your IT management gets more complex.

What started as a simple increase in the needs of your application, has changed into the growth of a complex IT organization.

An organization that was once agile tends to change and migrate over time. It changes into either a robust or rigid organization—one that is afraid of and rejects change in order to keep the system stable and supported, or it changes into a fragile organization—an organization where every minor change risks breaking a larger system or process, limiting the ability to adjust and grow. This is illustrated in Figure 1-10.

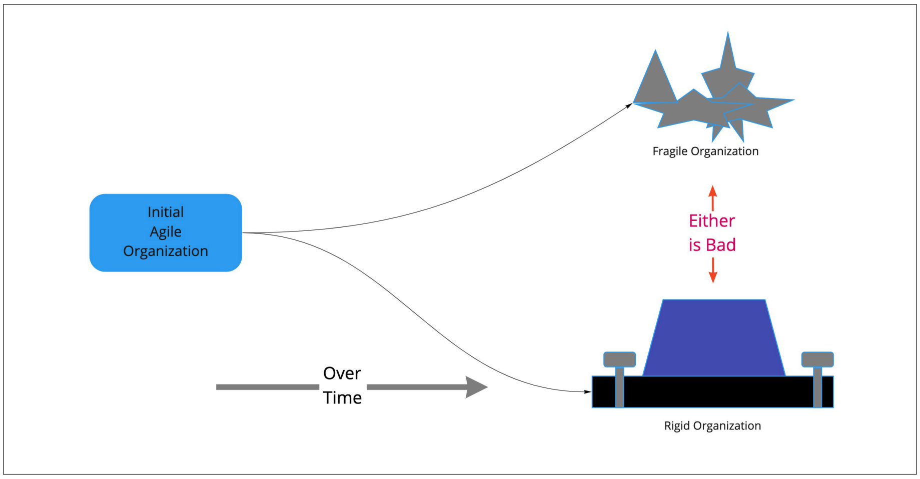

Figure 1-10. An Agile organization fails over time either by becoming rigid, or fragile.

As complexity increases, the ability of the organization to change direction quickly and respond to new demands diminishes. The organization becomes less agile.

As complexity increases, the ability of the organization to change direction quickly and respond to new demands diminishes.

Almost independent of the specific company, complexity and technical debt correspond to a lowered ability to respond to market and competitive demands.

Why is this true? There are typically two reasons why application complexity leads a company to lower agility.

First, when an organization has overly complex applications, changes to those applications are increasingly likely to cause problems. Small changes and small adjustments cause large failures and outages to occur. This makes the organization hesitate. Changes go through additional review cycles, changes get consolidated, changes that don't show clear value are discarded as too dangerous. Rather than having a "Yes, we can try that" attitude, the stock attitude of the organization changes to a much more conservative view. "No, not unless it's absolutely necessary" becomes the more likely answer.

This is the reaction to keep the application from failing. It's the company's response to keeping the application robust—the fewer changes you make, the safer the application is.

The application development process slows down considerably. This makes the application—and the company—significantly less agile than its competitors.

Second, if the organization doesn't naturally slow down, then it might continue to make changes. However, because of the application complexity, the organization tends to make changes without understanding what's involved in the changes. This risky behavior leads to dangerous changes, and these changes break things. The organization doesn't slow down and become conservative, but moves forward recklessly. The result? Application availability suffers and customer issues increase. Technical debt increases even more than it had previously. This feeds into the complexity and it creates a vicious circle. Complexity leads to brittleness which leads to failures which lead to technical debt which leads to complexity.

IT Death

So, technical debt leads to complexity, and complexity leads to organizational pain. This all ultimately leads to *IT death.*

But what does IT death look like?

IT death is what happens to an organization when the pain of complexity sends the organization into a state of ineffectualness. It cannot improve, it cannot grow, and hence it stagnates. Since competitors will continue to grow, an organization's stagnation ultimately leads to its death.

You see it in many organizations.

Xerox, long the leader in copiers for larger organizations, suffered from the inability to pivot from copiers to the personal computer. Despite the fact that Xerox PARC originally conceived the modern personal computer user interface,[1] they were unable to compete with Microsoft and Apple for the personal computer operating system. Arguably, without Xerox PARC, there would be no Apple Macintosh computer, yet Xerox's inability to pivot kept them from this innovation.

It's not just technology companies that suffer this fate. Firestone, the tire company, was facing the difficult task of modernizing its tire creation process in light of radial tire technology created by one of its competitors, Michelin. Firestone bogged down and could not update its processes to handle the new technology. Try as it might, it kept making tires that customers did not want, and their business suffered. Ultimately, Firestone was absorbed by Bridgestone. This is an example of what the Harvard Business Review[2] calls *Active Inertia*.

Many other originally highly innovative companies fall into the trap of IT death by losing their ability to innovate. Hewlett Packard, one of the founding companies of Silicon Valley—the heart of technical innovation across the world—found its lack of innovation lead to a slow death spiral.

And let's not talk about the innovation failure of Polaroid, which couldn't innovate new camera technology; or Blockbuster Video, which failed to embrace the importance of video streaming technology.

And Borders book store, which was overwhelmed by the innovation of the upstart Amazon.com.

Technical debt and complexity slow down innovation. They keep companies from staying competitive, and ultimately this results in their eventual downfall, and potentially even death.

What makes a mature IT organization

A mature IT organization is an agile organization. It can make decisions quickly and easily, stick by those decisions until organizational needs dictate a change, and implement those decisions quickly and effectively.

Why is it important for a mature IT organization to be agile? Without agility, companies fall into two traps that can bring them down:

- Lack of *competitive offerings.*
- Unsafe *security vulnerabilities.*

Let's look at each of these in turn.

Competitive Offerings Maintaining competitiveness is critical to a modern application. This is because the pace of change is accelerating. Technology is advancing and new competitors are emerging constantly. Your competitors are moving faster than ever before. If you can't keep up with your competitors, you quickly fall behind and soon become irrelevant. Keeping up means moving faster and faster, which means being able to adapt and change as the situation demands.

Customers are constantly pushing the feature, price, quality envelope with new and innovative ways of doing business. If you have a customer that is pushing you on price, quality, or feature set, you need to be able to respond to remain competitive.

New ideas are the lifeblood of a competitive company. When a new idea comes up, you need to be able to quickly adjust and adapt to enable the new idea. This requires agility.

Customers are looking for innovation when it comes to making a buying selection. Companies that appear innovative are more likely to get the customer's business. This means you need to respond to customer requests and customer needs faster. Failing to do this will not only lose you customers, but it will soon cost you credibility in the industry.

Agility is essential to maintaining a thriving business.

Security Vulnerabilities Your competitors aren't the only ones innovating. Bad actors are innovative as well.

Never before has the IT infrastructure of our valuable applications been at risk to security vulnerabilities and actions of bad actors as it is today. Bad actors are not only growing in numbers, but they are growing in sophistication as well. Bad actors are just as innovative at coming up with new ways to attack your application, as your competitors are innovative at coming up with new ways to attack your business success.

Bad actors are constantly innovating, improving their attack vectors, and exposing the vulnerabilities of our applications.

You, as a company and the owner of your application, must innovate constantly to keep your applications safe and secure. You have to constantly strive to keep one step ahead of the bad actors. This innovation requires agility.

Summary

Hence, the IT complexity dilemma. IT agility is critical to building a successful company, yet the very success itself adds technical debt and complexity, and this complexity leads to either rigidity, or fragility. In either case, ultimately, competitors will outpace the organization in innovation, and the organization dies. Long term success for a company means managing the IT complexity dilemma.

1 Xerox's Palo Alto Research Center.

2 Why Good Companies Go Bad, Harvard Business Review.

Auditing and Assessing Your IT Ecosystem

How do you avoid the IT complexity dilemma? First things first. To deal with IT organizational complexity or the complexity of your application you need to understand what makes up your organization.

So, the first step in understanding complexity is to perform an audit of your application, your IT organization, your teams, your delivery and operations processes, your company as a whole, and assess your current situation to determine what parts of your system contribute to your excess complexity.

Auditing vs. Assessment

Auditing and assessment are two distinct processes that are often used somewhat interchangeably to describe the process of understanding the components that make up a complex system, such as an enterprise application. But what's the difference between the two?

Auditing is typically the process of creating a *controlled inventory*. In this context, the word "controlled" is a governance phrase.

- A bank can count its money, but it has its records formally reviewed by an independent agency for accuracy when it undergoes an audit.
- A company keeps their own financial records, but if they are about to be acquired or merged, their records are *audited* independently to ensure the records are accurate.
- If you live in the United States, you keep track of your own personal finances and you submit records to the IRS every year to specify how much income tax you owe. Occasionally, the IRS can *audit* individuals to validate that the information they are providing is correct and accurate. Other nations have similar processes.

Whether it's a formal audit by an independent agency, an in-house compliance audit, or even an informal review, the word audit has a formality and control aspect associated with it.

In an IT audit, we are talking about determining, formally or informally, the components of our applications, the infrastructure they are running on, and the systems and processes they utilize.

Large enterprises may invest hundreds of thousands of dollars and spend six or more months creating such an audit, using a formal outside auditing firm. Or an architect may create a quick diagram in Visio, print it out, and put it in a log book. Both are essentially audits, but the former is much more formal than the latter.

Assessment is what happens next. Once you know what components make up your systems and applications, understanding how they work together, what each component is used for, why it exists, how it's important, and the overall impact on the system as a whole, is the assessment. Often, this is done as part of the audit, but it doesn't have to be. The word assessment also has connotations, just like auditing does:

- A teacher creates a test as an *assessment* to see if their students understand the material they were taught.
- A coach will evaluate how an athlete performs to *assess* how they can utilize the athlete in a team setting.

- A voter will *assess* the pros and cons of each side of an issue before casting a vote.

Assessing often implies grading, scoring, or evaluating.

Auditing and assessing are two sides of the same coin. They are complementary processes that work together to determine the makeup of a large, complex, enterprise application infrastructure.

When applied to IT applications, auditing and assessing are more akin to a *survey*. What is a survey? A survey is a view of the structure and architecture of our system that is built and maintained external to the system.

A survey is a form of *measurement* indicating how our application, infrastructure, business, or system operates and how it's structured.

What Do You Measure?

In business, including application development and IT infrastructure, our measurements are built around people, processes, and technology.

People

Do we have the right skill sets in the right places to allow our business to function successfully? Are our employees engaged and satisfied? Are we utilizing our people most effectively?

Processes

Do we make good and timely decisions using the right data? Are our business processes efficient and effective? Do we waste time or resources inappropriately?

Technology

Do we have the right applications for our business to function? Are those applications running in the right infrastructure? Are we properly utilizing all aspects of our infrastructure? Do we waste technology we have purchased by not using it effectively? Are we inefficient because we have not acquired a piece of technology that may help us?

Determining what to assess is important but highly case dependent. Focusing on questions, such as shown above, is a good way to brainstorm on what to measure. This gives you a great perspective on the sorts of things to measure when doing an assessment and audit.

Why Do You Measure?

You can't track how your organization progresses in its growth without understanding the state of your organization and how it's currently functioning. Only by measuring can you know what components make up your system, and understand how they work together.

> *To determine where you can improve, you must measure where you are currently.*

MEASURE-TRY-MEASURE-REFINE

Before you make any change to how a system operates, you want to determine how that change might impact the system. In order to do that, you need to measure the current state of the system before you make a change. Then, when you try making a change, you can re-measure and assess the impact. This allows you to refine your attempt and measure again. The result is a loop, called the *Measure-Try-Measure-Refine* loop, illustrated in Figure 2-1.

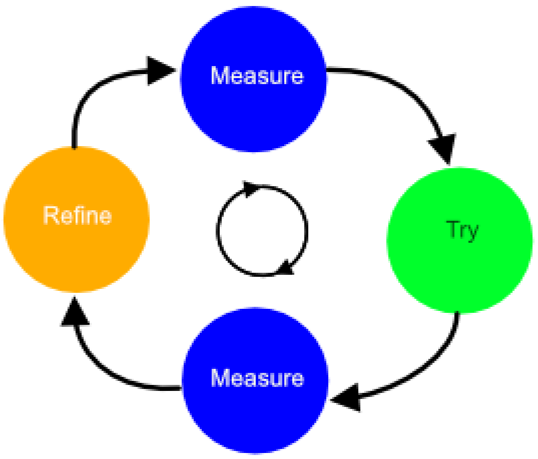

Figure 2-1. Measure-Try-Measure-Refine Loop

This is a basic process of cyclic improvement, and it goes by many other names. It's very similar to the Plan-Do-Check-Act (PDCA) or Plan-Do-Study-Act (PDSA) cycle, otherwise known as the Deming cycle[1].

In this version, we start at the top with *Measure*. We measure our system to understand our current state before making any changes.

We then move on to the *Try* step. Here, we attempt a change to see whether it positively or negatively impacted our system—did it help us or hurt us.

We then do another round of *Measure*. We measure our system again to see how it has changed from the original measurement. By comparing our current state with our original state, we can determine if what we tried improved our situation or made it worse.

We then *Refine* our attempt to account for the problems and expand on the benefits. The net result, we should be in an improved situation.

This leads us back to the top where we *Measure* again and start the cycle all over again, for continuous improvement.

Measure-Try-Measure-Refine. Then repeat.

This is the process of continuous improvement.

We must measure before and after we make any change and determine the difference between the two measurements, to recognize the impact of our attempt.

You can't tell if you improved unless you know your situation before and after every change you make.

Every change has a cost associated with it. Costs may be tangible costs such as engineering costs, testing costs, operational impact of a change, or less tangible such as opportunity costs. Sometimes we make a change that improves our situation, but the cost is greater than the value of the improvement. We may have made a *localized improvement*. But overall, we are not better off because the cost was too great. Only by measuring can we understand the real value of the change we made as well as the associated cost.

1 PDSA Cycle, The Deming Institute. https://deming.org/explore/pdsa/

Another benefit of measurement is knowing when you are done. This cycle could go on forever, but sooner or later the improvements you make are no longer worth the cost of making them. We can use our measurement to know when to stop. When have we reached our goal or when have we made as much improvement as we can without incurring unreasonable cost or burdens. This process stopping based on measurement is illustrated in Figure 2-2.

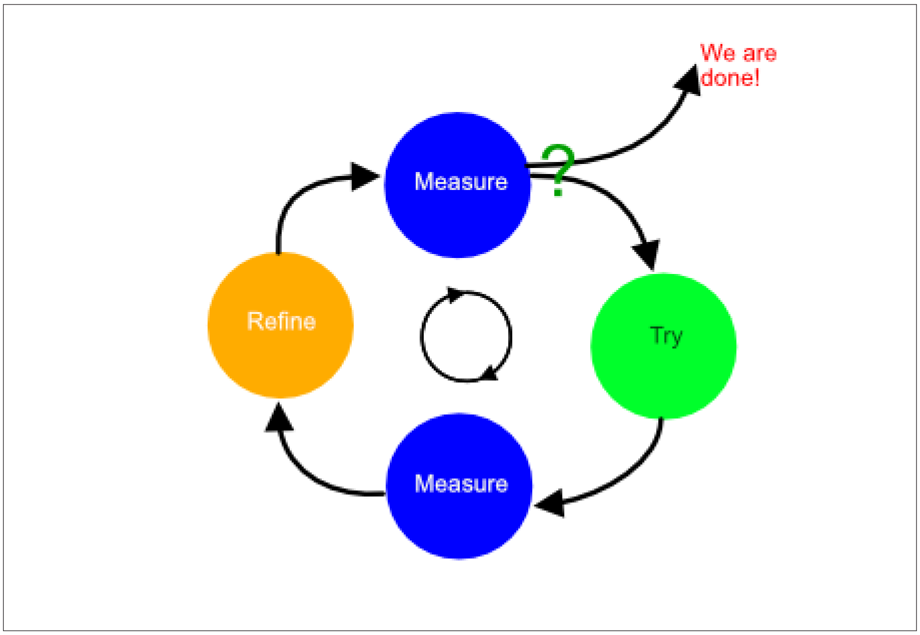

Figure 2-2. Measure to determine when it is time to stop

In summary, measurement helps us:

1. Determine our current system's state and where we are in our process. So we know where to place our energies to improve.
2. Analyze our changes to see if they've made things better or worse, or better enough given the cost of the change.
3. Determine when it is time to stop making improvements.

How Deep Do You Measure?

A large enterprise may spend six months on a large, formal audit of their system. This is a form of measurement. It will give an accurate state of the system that is complete and highly detailed, but will show the system as it was six months ago, when the audit started.

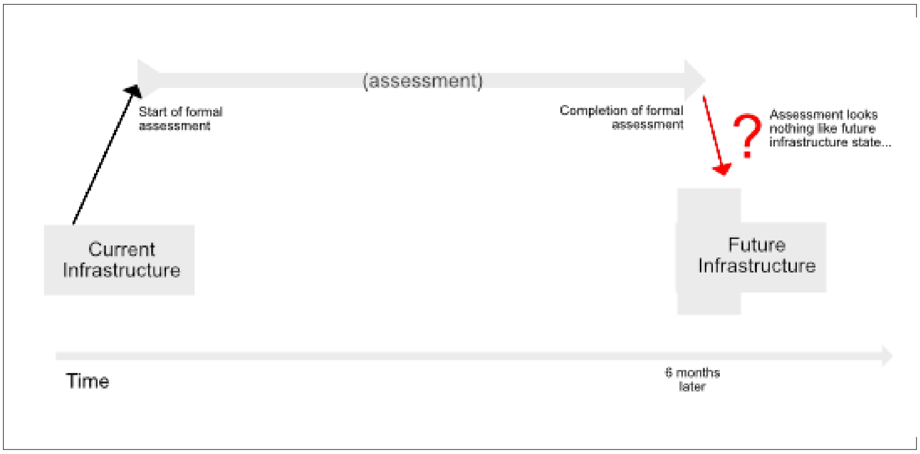
(assessment)
Start of formal assessment
Completion of formal assessment
?
Assessment looks nothing like future infrastructure state...
Current Infrastructure
Future Infrastructure
Time
6 months later

Figure 2-3. Formal assessment is inaccurate the moment it's completed

Such an audit is not of any use to us in managing systemic improvements, when we are trying to incrementally improve between each measurement cycle. If we can only make small changes every six months, after the end of a formal audit/measurement cycle, then we can't make changes very fast and our entire Measure-Try-Measure-Refine process fails. In order to succeed at getting out of the IT Complexity dilemma, we need to get out of unwieldy long and expensive evaluation cycles. Our Measure-Try-Measure-Refine cycle duration should be measured in days or hours, not in months or years.

So, when you create an inventory of your system, when you create a review of your applications, when you *measure* the current state of your application in its infrastructure, how deep should you measure the system?

One viewpoint says that you need to have a deep and detailed view into your application and its infrastructure. You need to know about every CPU, data memory, network segment, cable, application procedure, service, node, etc. Without knowing everything, you know nothing. This viewpoint is the approach that creates the six month formal external audits we talked about previously. By requiring precise measurement of everything, you can't possibly know everything you need in a timely manner. It can also lead to knowledge without understanding. By the time you have measured everything, you have no context for applying the understanding gained.

I was big into computers as a kid, and living in the midwest where storms (thunderstorms, tornados, and snowstorms) were a big deal and it generated an interest in meteorology. In the 1970's, neither of these technology fields (computers or meteorology) were very advanced. Computers were simple and just starting to be understood, and meteorology was just starting to leverage computers and satellite technology in a serious way.

At that time I was told by someone who was an expert in meteorology that "We now have the ability to completely understand what the weather will be in any given location 24 hours in advance." At the time, this was an amazing comment to hear. Meteorology wasn't anywhere near that precise back in those days, but this person was saying we could predict the weather.

But, this expert went on to say, "the problem, though, is it'll take our fastest computers 3 days to figure out what tomorrow's weather will be." It's not very useful to figure out what tomorrow's weather will be, if it takes three days to figure it out.

Thus was my first lesson in the concept that *too much data can be worse than not enough data*.

Sometimes, too much data isn't helpful, especially if the cost of getting that data makes the data inherently less useful. If you need six months to collect the data, you can't use that data to determine what changes you need to make *today* to make things better *tomorrow*.

Instead, you will need to compromise. You'll need to collect some sort of subset of data, with the expectation that the subset you collect gives you the insight you need to extrapolate the rest of the data. This is illustrated in Figure 2-4.

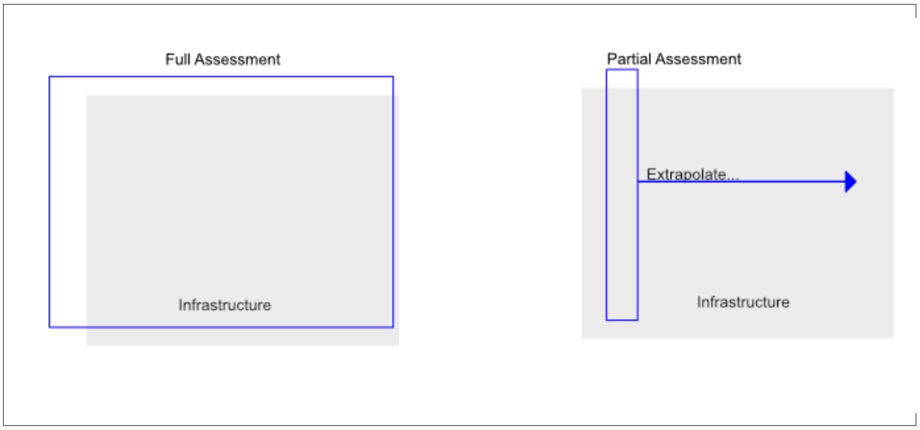

Figure 2-4. Rather than doing a full assessment, you can do a partial assessment and extrapolate the results

But what data do you need? Let's assume you have a large system that is running many large applications and you want to do an inventory that lets you know how many infrastructure components are needed by each application. You are doing this so you can compare these numbers to the amount of inventory you actually have on hand and understand if you have excess capacity, or are running your services too lean. But taking a complete inventory will be too expensive and take too long. How do we determine what we have without doing a full inventory of our entire system? Let's look at some possibilities.

Option 1: Look at only some attributes of our inventory (such as compute) and ignore others (such as networking or storage).

One subset you can do when you create your inventory is to only look at the CPUs you have in your infrastructure. How many CPUs do you have? How powerful are they? How much raw computation power does that represent? You can then determine how much computation power you have assigned to each of the parts of each of your applications.

This type of inventory only focuses on raw computation power. It ignores other important aspects of the infrastructure an application requires, for example memory, storage, and network. The extrapolation you make is that, if one application has twice as much computation as another application, then you can assume it also has twice as much networking, memory, and storage.

However, this is rarely accurate. Just because an application has twice as many CPUs assigned to it, or twice as powerful CPUs, does not mean that those CPUs have twice as much memory, or that you have twice as much database storage, or that you have twice as much networking capacity.

Using computation as a proxy for your entire system inventory leads to inaccurate data and bad expectations.

Option 2: Look at a single application or service in its entirety, but ignore all other applications/services.

Another subset you can look at when creating your inventory is to look at one single application or service in its entirety. You determine how much computation it requires, how much memory, storage, networking, etc. You figure out what services compose this application, and what external services this application requires to operate. You make a complete inventory of everything required to run that one particular application.

Then, you use this information, and make the assumption that all the other applications or services in your system will have a similar set of requirements. If your analyzed application needs 25 servers, and another application needs 50 servers, then you can assume that the other application uses twice as much *everything* as your analyzed application.

This is perhaps a bit more accurate than option 1, but as you can imagine it is still inaccurate. How much infrastructure a given application or service requires does not have much bearing at all on how much another application or service requires. Two applications will look very differently internally, and they will use a very different mix of support services. You are still no closer to understanding the complete infrastructure inventory for your entire system.

Neither of these options is a very useful model for getting a complete look at the inventory of your application.

Then, how do you build your inventory short of doing a full, multi-month audit of your entire infrastructure?

Each of the above options created inaccuracies while speeding up the inventory by pulling out important data from your dataset. Rather than pulling data out of your dataset, how about simply spending less time creating the inventory list in the first place? Rather than trying to accurately and completely create an inventory of a small section of your infrastructure, let's try and create an overview inventory that may not be very accurate, but will represent your entire system.

ADAPTIVE ASSESSMENT

Let's accept the fact that this may not initially be a very accurate inventory. Instead, as time goes on and we adjust our system, let's add more data to our existing, incomplete and inaccurate inventory, and little by little build up the quality and accuracy of the resulting inventory. As time goes on, our inventory will be adjusted and changed, and at each change, we add more data to our inventory and make it a more accurate overall assessment of our system.

I refer to this as an *adaptive assessment*. Rather than striving for perfection in the inventory—either the complete inventory (which will take too long) or in a part of the inventory (which will be unrepresentative)—we strive for an *estimate* of the inventory for the entire system. Then, over time, we'll refine the estimate and make the estimate more and more accurate.

It's an iterative approach to inventory assessment.

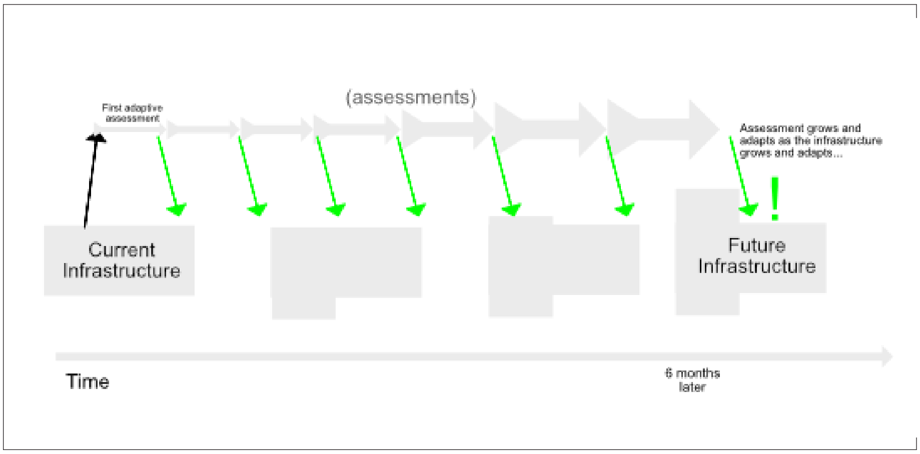
First adaptive assessment
(assessments)
Assessment grows and adapts as the infrastructure grows and adapts...
!
Current Infrastructure
Future Infrastructure
Time
6 months later

Figure 2-5. An adaptive assessment grows and expands with your infrastructure

Figure 2-5 illustrates this adaptive assessment process. In the adaptive assessment, you are trading some amount of inaccuracy for speed. You are exchanging *granularity of assessment* for development speed.

Realizing your inventory will be worthless if it takes six months to finish (because the inventory you have six months from now will not be the same inventory as what you have now), you instead create a shallow assessment of your inventory, and refine it over time. You will have a much more relevant and useful inventory for a greater period of time. You will get your initial, albeit inaccurate inventory very quickly. As your system changes, you strive to add more detail to your assessment. Little by little your assessment becomes more accurate. Because the inventory didn't take much time to create in the first place, it stays as accurate as possible to your *current state* rather than a six month old *historical state*.

Note

An *adaptive assessment* trades granularity for speed. Since an assessment of where you were six months ago is irrelevant, a less refined estimate *now* is better than an accurate assessment *later*.

Value of an Adaptive Assessment

The true value of an adaptive assessment is time. We will begin to talk about adaptive organizations in the next chapter. But an adaptive organization with an adaptive architecture requires decision making cycles that are measured in days or weeks or less. In many cases, the action execution cycles are measured in minutes or days. Since a formal audit can take weeks or months, it's useless and wasteful to an adaptive organization. Waiting six months for the results of a formal audit is just not practical or often even useful, and even if you could wait the six months, you would have a six month out of date view of your system. If you make decisions and changes daily, the assessment will be woefully out of date months before you ever see it.

An adaptive assessment gives you *some* actionable information immediately. Even if you don't have full access to everything you would have available in the formal audit, what you do have available to you is available at the right time. Some data when you actually need it is infinitely better than complete data when it's too late to be useful.

Some data when you actually need it is infinitely better than complete data when it's too late to be useful.

The key is to keep the end goal in mind. Ultimately you do want to end up with as accurate an assessment as possible—both due to the quality of the information and the accuracy of the information in time (an inventory that represents the current state of the system, not some point in time view of the system in the past).

The trick then is this: how do you decide what level of granularity you are willing to lose in your assessment in order to have the data you need in a timely manner? Put another way, how granular is still useful and when do you need it by?

Creating an Adaptive Assessment

Start with this optimistic goal:

We want everything.

But we need to temper that goal with a reality check:

I am willing to accept errors in what we collect.

We may end up with guesses and estimates in much of our assessment in order to get our results more quickly.

We will be using the **error bar approach** to an adaptive assessment. This approach results in a complete assessment—a complete inventory of our system. But our assessment is based on guesses and estimates, which may be wrong. Hence they have “error bars”. Our goal is, over time, to revise our estimates as we gain more information in order to reduce the size of the error bars, ultimately creating a more accurate assessment, such as the right hand side of Figure 2-5. We strive for a complete inventory, with errors, quickly. This is opposed to the extrapolation approach which strives for a partial inventory, quickly. Or the formal audit which strives for a complete inventory, without errors, after a long time.

A classic error bar approach starts with colloquial knowledge about the system. “I believe we have 48 network access ports in our primary service rack”. Close enough, we’ll use that number for our inventory for now. At some point in the future we can verify that number. If we find it’s 50 instead of 48, we can adjust our assessment and we will end up with an assessment that is eventually more accurate. But, in the meantime, we have a workable number we can deal with. Knowing that we have “around 48 ports in our rack” is infinitely more useful than saying “I don’t know how many ports we have, and we won’t know for several months”.

Adaptive assessments say that “I value being educated sooner rather than later, and I understand I will get more educated as time goes on”.

Decisions Based on Adaptive Assessments

Of course, we will be using our assessments to make decisions. It's important to understand that our assessments are not absolute. They are estimates, and will be adjusted and refined over time. Our data—our assessment—is **agile**.

As a result, the decisions we are making using this data need to be agile as well. We need to be willing to rethink and reimplement decisions when we are presented with more accurate and more refined data. This does not mean we can flip-flop on decisions routinely.

Decision flexibility is a struggle for many IT organizations. Some organizations cannot change directions easily, even with clear and compelling evidence they are headed in the wrong direction. Meanwhile, other organizations can't stay focused on anything and constantly move back and forth in a series of non-decisions. Neither approach is a good place to be.

To effectively use adaptive assessments, you need to be willing and able to make decisions that stick, but be willing to rethink those decisions *when and if the refined data you have available suggests a change.*

Your data has error bars, so your decisions must have error bars too. Flexibility is important, while still making actionable decisions.

Your data has error bars, so your decisions must have error bars too.

The most important skill you need as an organization is adaptivity. Your architectures need to be adaptive, not overly robust. An architecture that is rigid and resistant to change is not an architecture that is suitable for adaptive assessment.

Loose Coupling

Many architectural patterns support adaptive assessments, but one of the most valuable patterns that you should embrace in all your architectural decisions — application, infrastructure, business — is the pattern of a **loosely coupled architecture**.

Loosely coupled architectures are where the connection between any two modules within the architecture is as loose and flexible as possible. Loosely coupled architectures apply to application architectures (such as service oriented architectures), infrastructure architectures (such as cloud centric architectures), and to business and organizational architectures.

For application architectures.
: In the case of application architectures, loose coupling means you must create solid APIs and contractual agreements between software services that define the expected black box interactions between the services. These solid APIs create and manage the inter-service expectations. But they do not put any requirements on the actual methods, systems, and architecture used in the internal implementation of the services themselves.

For infrastructure architectures
: In the case of an application infrastructure, loose coupling means you should depend on using *infrastructure services*, such as *cloud-based services*, that have predefined expectations about how they work. However, the user of these services does not need knowledge on how the infrastructure service itself is actually constructed or how it functions.

For business and organizational architectures
: In the case of organizational architectures, loose coupling means defining the ownership and responsibilities of individual teams, independently from the ownership and responsibilities of other teams or the organization as a whole. Teams have clearly defined goals that are achievable independently, without requiring undue intervention from neighboring or connecting teams.

In my O'Reilly Media book **Architecting for Scale, 2nd Edition, you can read much more about loose coupling**. This book has five tenants for building highly scalable applications and organizations, several of these tenets describe patterns that involve loose coupling. Tenet 2 is about loosely coupled applications (service oriented architectures). Tenet 5 is about loosely coupled infrastructures (cloud based architectures). Tenet 3 talks about loosely coupled organizations and processes. And Tenet 4 talks about risk management, which is an important part of building and using adaptive architectures as a whole.

Examples of Adaptive Assessments

There are many ways to perform adaptive assessments. How do you get started? Here are a couple examples that apply to IT infrastructure assessments.

EXAMPLE 1: THE BRAINSTORM ADAPTIVE ASSESSMENT

An adaptive assessment can start with nothing more than a group brainstorm to try and write down what the parts of your infrastructure are, and how they work together. The result may not be very accurate at all, but it is still a useful assessment of your system, because it gives you a view of how people *think* the system is constructed. As you find mistakes and correct errors, you'll grow that understanding and you'll be able to refine your team's internal understanding of how the system *actually* functions. You should encourage your teams to update the assessment every time they interact with the system and make changes, so the assessment continues to improve over time.

Your brainstorm is an adaptive assessment because it meets the two core requirements:

1. **Quick results**. It generates results at some level of quality very quickly. After the initial brainstorming meeting, you have an assessment. It may not be accurate yet, but it's a start and has immediate value.
2. **Improve over time**. The results improve as time goes on. As your team keeps updating the assessment as they make changes, the assessment keeps improving.

EXAMPLE 2: THE CLOUD TAG ADAPTIVE ASSESSMENT

A convenient way to start an assessment of your operational infrastructure, when using cloud architectures is to begin by virtually tagging individual components of your infrastructure. This is particularly useful in cloud based systems as most cloud providers give you the tools necessary to tag individual infrastructure components. By starting to tag infrastructure components with specific attributes, you can begin to see how you are utilizing your cloud infrastructure. You can tag infrastructure components to show which applications use a specific component, what teams are responsible for managing it, who is responsible for paying for it, and who to contact to determine when the component is being used, or if the component is still being used at all.

Once you've started the process of tagging your cloud infrastructure resources, you can generate reports based on those tags to find out all sorts of useful things. Who owns which components, what team uses which services, who uses excessive resources, who has spare resources, and who is running their resources too hot.

By putting policies into place requiring all new cloud components to be properly tagged and encouraging teams interacting with existing cloud components to add proper tagging if they do not have them yet, you'll keep improving your assessment of how your cloud infrastructure is working. Your assessment will improve over time.

Eventually, you may even want to utilize a cloud service that will enforce tagging rules. Some enterprises setup polices that ensure tagging by simply systemically deleting resources that are not tagged correctly. Nothing will encourage a team to make sure their cloud infrastructure resources are properly tagged more than having a critical infrastructure component simply disappear from their application because it wasn't tagged correctly!

Your cloud tagging assessment is an adaptive assessment because it meets the two core requirements:

1. **Quick results**. It generates results at some level of quality very quickly. Simply tagging a few very visible resources will give you some level of reporting ability.
2. **Improve over time**. The results improve as time goes on. Every time you create a new cloud resource going forward, make sure to tag is appropriately (software can actually require this task before it creates the resource). Existing untagged resources are tagged as they are noticed. As time goes on, a greater percentage of resources will be properly tagged.

The Survey Analogy

At the beginning of this chapter, I said the word *survey* is a better description of what we are doing, than the words *auditing* and *assessing*, which are historically used. The process of an adaptive assessment is more accurately described as a survey from another perspective as well.

Think about using surveys as a way to get information about people. Politicians do surveys all the time. Companies with visible brand loyalty use surveys to understand the value of their brand. And employers use surveys to. In fact, employers frequently use surveys to determine how their employees are doing. They do what are called *employee satisfaction surveys*.

Let's think about an employee satisfaction survey for a minute. When we conduct one of these surveys we get is a list of what percentage of respondents gave specific answers to the specified questions. That's all we get. This is not the same thing as whether employees are happy or not. Instead, we *deduce* whether our employees are happy or not from the answers. This is an assumption based on the data.

As time goes on, we can repeat the survey. We can even fine tune some of the questions to get better and more accurate results. By comparing results over time, we can get a more accurate answer to the question about whether our employees are happy or not, as well as whether the things we are doing are improving our employee happiness.

These surveys are non-IT examples of *adaptive assessments*.

Summary

An adaptive assessment is a highly effective way to get an audit/assessment of your business, processes, applications, and infrastructure quickly and effectively. It assumes that *some data, even if not 100% accurate* is better than *no data at all*. An adaptive assessment is characterized as an assessment that: 1) generates quick results and 2) improves the quality of those results over time. A successful adaptive assessment requires both of these characteristics.

Moving to an Adaptive Architecture

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

One of the greatest tools that the modern cloud era has brought us is the development and growth of the adaptive architecture. The adaptive architecture helps IT organizations build applications and systems that are more flexible and hence more agile. Used properly, adaptive architecture is the leading component in decreasing IT infrastructure costs associated with cloud computing. Finally, and perhaps most importantly, adaptive architectures are a major tool in the process of reducing IT complexity and hence technical debt.

But what is an adaptive architecture?

Adaptive Architectures

An adaptive architecture is any architecture design that is able to be changed dynamically and programmatically without the need for physical intervention and manual operations in the change.

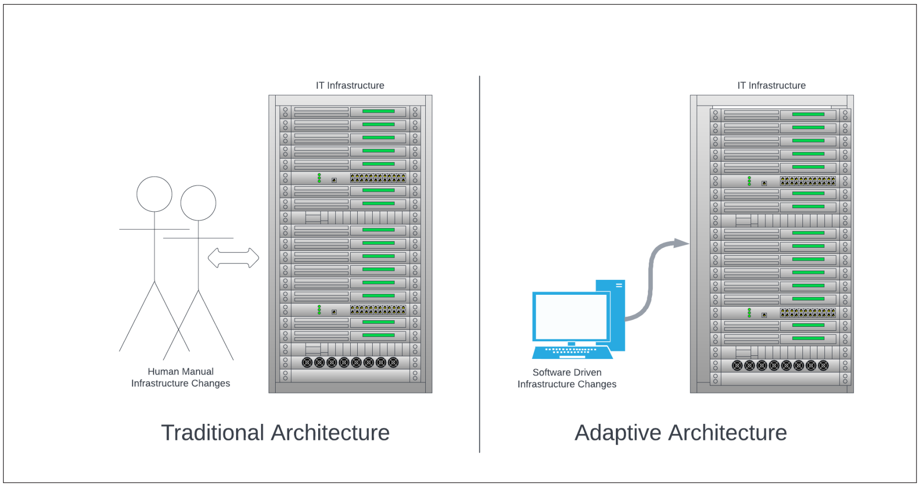

Figure 3-1. Traditional vs Adaptive Architecture

The left side of Figure 3-1 shows a traditional architecture. Human workers manually work with the individual infrastructure components — servers, switches, network cables, etc. — and make changes as needed to adapt to the growing needs of the organization. If additional server resources are required, someone manually goes in and adds a new server to the rack, wires it up, and gets it up and running. This process could take days, weeks or even months in some cases. This is opposed to the adaptive architectures, such as those provided by cloud computing providers. In these architectures, a software program determines the needs and dynamically, in real time, changes the IT infrastructure allocated to an application, and changes how it is configured, all automatically without human interference. If a new server is needed, one is allocated from the cloud service provider and attached automatically at the correct location within the infrastructure. Changes that took hours or days in the traditional architecture world take minutes or seconds in the adaptive architecture world. This allows architectures to be constantly and creatively modified on the fly to meet the ever-changing application needs.

If a service is getting additional traffic and needs some additional resources to handle the traffic, they can be automatically and dynamically added as needed, then freed when the need no longer exists. Need to perform a long complex operation, such as calculate a monthly report or process a new dataset? Temporarily allocate the additional resources, configure them as necessary, and perform the operations.

Want to try out a new idea in the running application? Temporarily allocate the additional resources to allow you to try the idea. If it works, you can put the resources into the IT infrastructure set. If it doesn't work, simply delete them and they go away.

Want to see how your new feature will perform under a heavy production load before launching it in production? Spin up some servers to run the feature and spin up some servers to generate fake traffic and give it a try. When you are done, simply delete the unneeded resources.

Is your application under a security attack and you need to reroute traffic around a compromised system? Try launching a replacement patch and pull the compromised component out of the infrastructure. Try to do that with a traditional architecture.

Adaptive Architectures in Action

Adaptive architectures give us flexibility and adaptivity. They allow us to try new ideas easily, add new resources to handle unexpected loads, and replace broken, failed, or vulnerable components quickly.

How do adaptive architectures fit into our modern application architecture? There are several places where they can assist.

AUTOSCALING

In most modern online applications, the resource needs of the application vary. Typically, they vary in relationship to the amount the application is currently being used, whether that's measured by the number of simultaneous users, the amount of processing going on for each user, or the amount of data being handled by each user, the general case holds true. The more an application is used, the more resources it takes to operate that application. This is illustrated generically in Figure 3-2. The more an application is used (horizontal line of chart), the more resources it consumes (vertical line of chart).

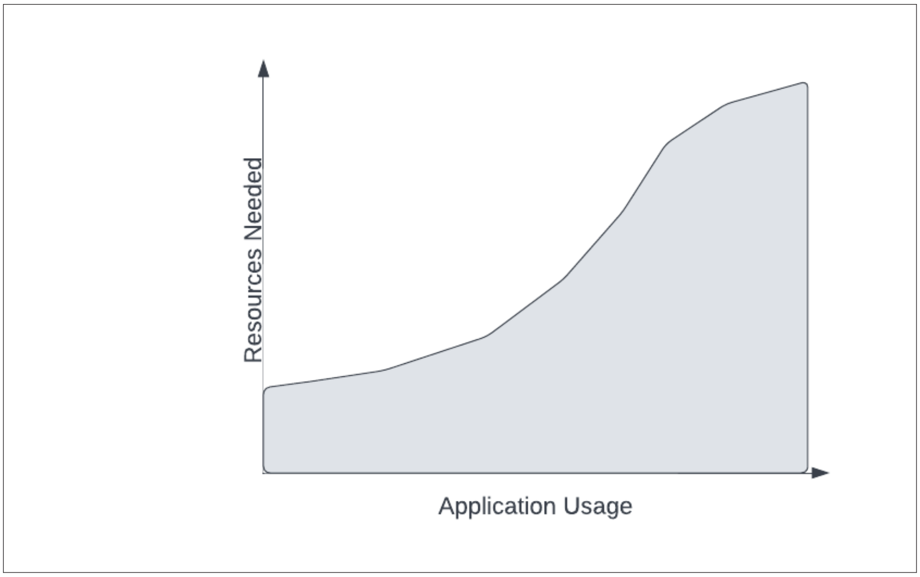

Figure 3-2. Resource requirements go up as application usage increases

In traditional application architectures, this creates a quandary. Resources need to be allocated to the application, but how many resources do you allocate? Given that historically (before the cloud), resource allocation required money and effort to set up, the resource allocation tended to be static. Given that resource usage is dynamic, that resulted in cases where either excess resource capacity sat idle for long periods of time, or the application ran out of resources due to insufficient capacity. Figure 3-3 illustrates this. If resources are statically allocated, they do not change as application usage changes. This means either you have excess capacity that is idle, or you have insufficient capacity and your application is starved.

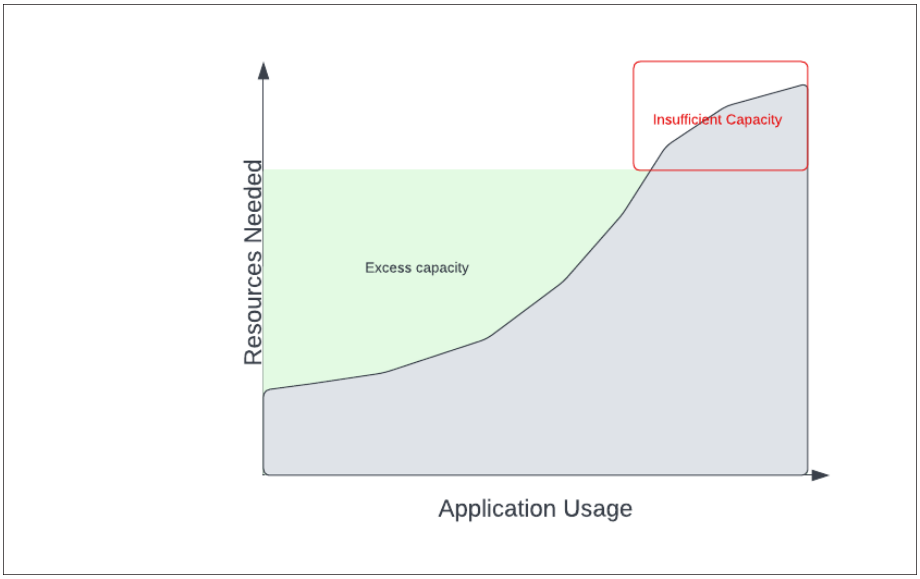

Figure 3-3. Static resource allocation results in excess capacity or insufficient capacity

However, with adaptive architectures, the quantity of resources allocated to an application does not have to be static. It can be dynamic — adaptive — and change based on the needs of the application. As application usage increases, additional capacity can be added to meet those needs.

This is illustrated in Figure 3-4. As application usage increases, adaptive infrastructure capacity adjusts dynamically to meet the needs. Ideally, you always have just a bit more resources available than the application requires, so you do not have significant excess capacity, nor do you ever have insufficient capacity.

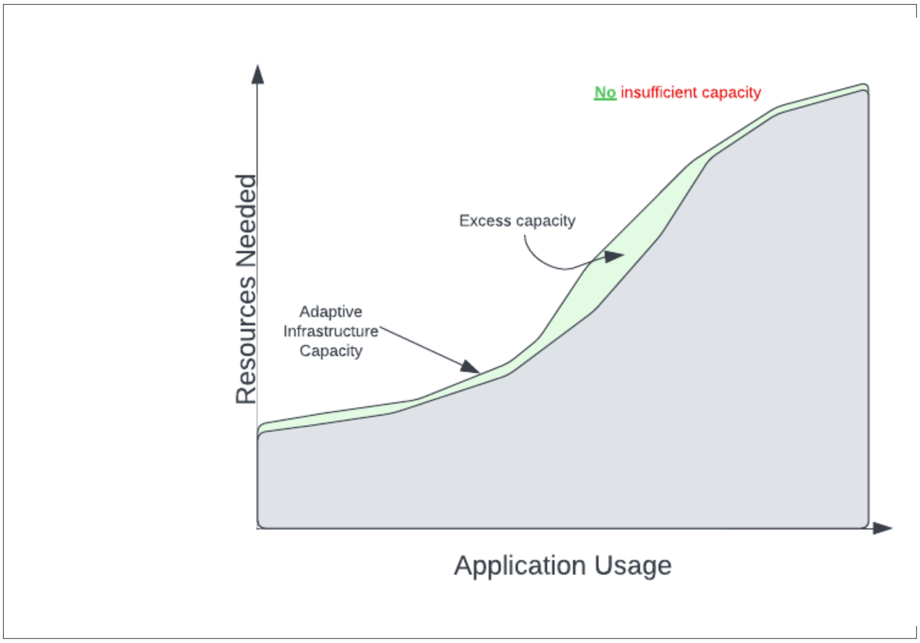

Figure 3-4. Adaptive architectures dynamically change allocated resources based on needs

By far, the best known example of adaptive architecture is *autoscaling*. Autoscaling is a cloud mechanism that monitors the resource requirements of an application, and adds or removes resources from the application as needed in order to meet the ongoing needs.

As an example, AWS has an Auto-Scale service which can dynamically change the number of servers involved in a server pool. It does this by monitoring various analytics of the server pool and determines if more or less capacity is required, and either adds or removes servers from the pool to meet the existing needs.

Some cloud based services do this allocation automatically and behind the scenes. For example, AWS Elastic Load Balancer will dynamically change the size and number of servers to a given application in order to handle the rate of incoming requests. As the number of requests increases, additional capacity is automatically added. That capacity is removed when the number of requests decreases. All of this is done completely transparently to consumers of the load balancer, it's handled internally to AWS's control systems.

SELF-HEALING

Imagine an application running on a pool of servers. What happens if one of the servers in the pool begins to fail or becomes damaged? This is illustrated in Figure 3-5.

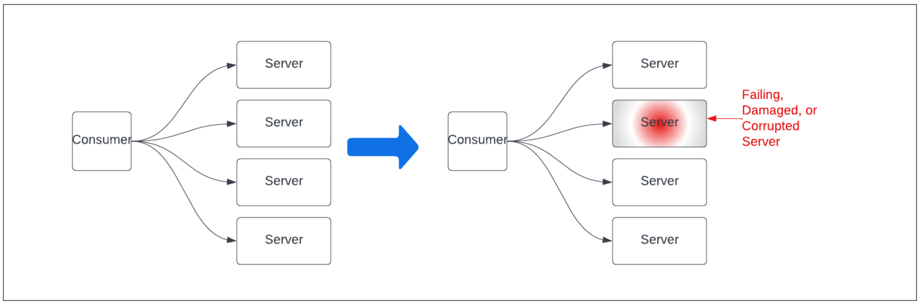

Figure 3-5. A server begins to fail in a server pool for an application

In a traditional, non-cloud application, your application will begin to operate sluggishly or may start failing intermittently. A human would need to get involved, login to the failing server, and repair whatever is going wrong with it. If processes have terminated, they need to be restarted. If a file is corrupted it needs to be repaired. These fixes may be fast, or they may be time consuming. If they are time consuming, the server may be removed from the pool temporarily while the repairs are performed, causing the application to operate at a reduced capacity.

In a cloud-centric application, adaptive architecture technology can be used to assist the repair process. If a server begins to go bad, rather than summoning a human to attempt to repair the server, instead the application can simply terminate the failing server and remove it from the pool. Then, allow the adaptive architecture technology to realize that additional resources are needed in the pool to run the operation, and a new server is spun up and added to the pool automatically. This is illustrated in Figure 3-6.

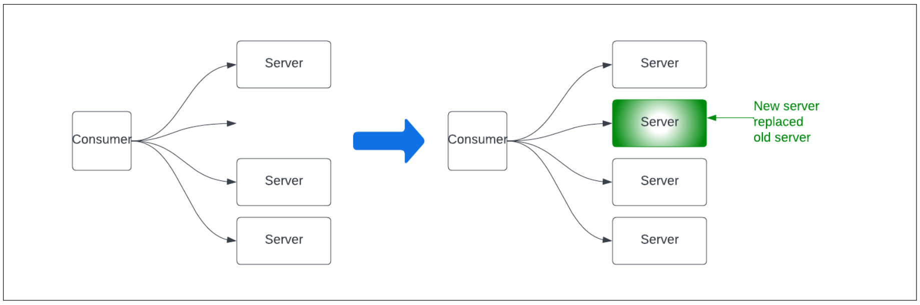

Figure 3-6. Adaptive architecture can spin up a replacement server

Rather than engaging a human to attempt to repair the ailing server, the ailing server is simply abandoned algorithmically, and a replacement server is automatically put into place.

There are two main advantages of this approach:

1. A human did not need to respond to the event. The problem was fixed automatically and dynamically.
2. Since a human did not have to respond and fix the problem to resolve the issue, the issue could be resolved substantially faster and potentially with less disruption to the users of the application.

Adaptive architectures can automatically repair failing infrastructure quickly and effectively without human interaction for many high profile and high impact failure modes.

INFRASTRUCTURE AS CODE

Traditional on-premises infrastructure is historically constructed by racking servers and other hardware together, connecting cables, and wiring up communications components.

This is a painstakingly manual process that is not only error prone, but also unreproducible and untraceable. It's unreproducible because of human involvement. There is no guarantee that one person will connect a computer in exactly the same manner as someone else does, and the result is a mixture of systems that are not consistent. Further, if a mistake is made and a problem is introduced, it's not easy to trace where that problem was introduced and hence how to solve it.

Adaptive architectures provide solutions to these problems. Since in an adaptive architecture your infrastructure is constructed programmatically rather than manually, you can create standard processes and systems to perform the same connections consistently and repeatedly .

This ability has led to a best practice for infrastructure creation known as Infrastructure as Code (IaC). IaC is an approach to automation that allows a description of the desired infrastructure setup to be created in a simple text document using a standard infrastructure description language. This document is then fed into an IaC system that issues the appropriate calls to the cloud and component APIs to construct a real cloud infrastructure that matches the documented infrastructure. This is illustrated in Figure 3-7.

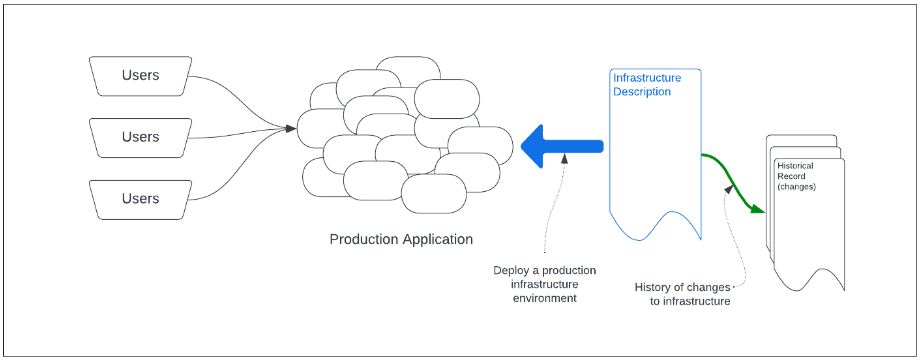
Users
Users
Users
Production Application
Infrastructure Description
Historical Record (changes)
Deploy a production infrastructure environment
History of changes to infrastructure

Figure 3-7. Basic Infrastructure as Code model for infrastructure management

Then, if a change is needed to the infrastructure, you simply make an adjustment to the document, and re-feed the document into the IaC system, which will issue API calls to adjust the infrastructure to match the updated documentation.

There are many advantages to this model of infrastructure management. First, the process simplifies the design of complex infrastructures and makes the process more approachable for an average software developer. This allows the development team to be heavily involved in the construction and management of the infrastructure that operates their software, allowing more consistent and efficient use of hardware resources, and enables better coordination of operations and development using DevOps principles.

Second, infrastructure documents can be managed just like software code can be managed by putting the document into a revision control system, such as *git*. This allows infrastructure documentation to go through review and approval processes, just like the software application itself uses. In fact, the exact same processes for software quality control can be used for infrastructure quality control.

Finally, if an infrastructure change causes a problem in the operating application, a review of the revision history can be instrumental in understanding where the problem originated and how to resolve the problem. This removes the troublesome quandary of trying to figure out “what changed on the server???”. All changes are explicitly tracked and managed, and can be reviewed as needed later.

DEVELOP IN A PRODUCTION-LIKE ENVIRONMENT

Infrastructure as Code (IaC) also gives you additional benefits. Since the infrastructure document describes an accurate view of the exact hardware setup required to build your production infrastructure, that same document can be used to easily setup auxiliary, non-production versions of the infrastructure. This includes setting up staging and QA environments, and ensuring they are identical in design to the production infrastructure. The common problem of divergence and infrastructure drift that often occurs between production and staging/testing environments is eliminated because all environments are created from the same source.

Additionally, a developer that wants to test a design in a production-like environment can easily *spin up* a production-like environment of their own that allows them to test their own changes safely and in isolation from both production, as well as from other developers and testers. This is illustrated in Figure 3-8.

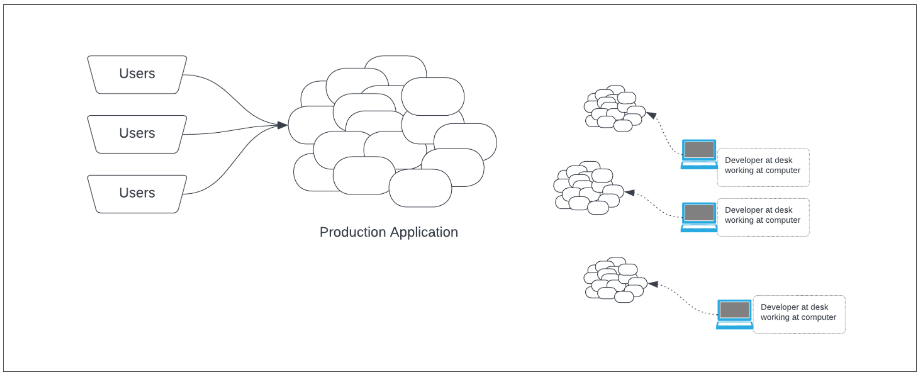
Users
Users
Users
Production Application
Developer at desk working at computer
Developer at desk working at computer
Developer at desk working at computer

Figure 3-8. Production application and equivalent development instances can all be identical and managed

Many identical copies of the production environment (perhaps identical, or perhaps scaled down in size but otherwise identical) can be created and managed easily using adaptive architectures and IaC techniques.

LOAD TESTING

The same IaC techniques can be used with adaptive architectures in order to create simulated environments for load testing. Take a look at Figure 3-9.

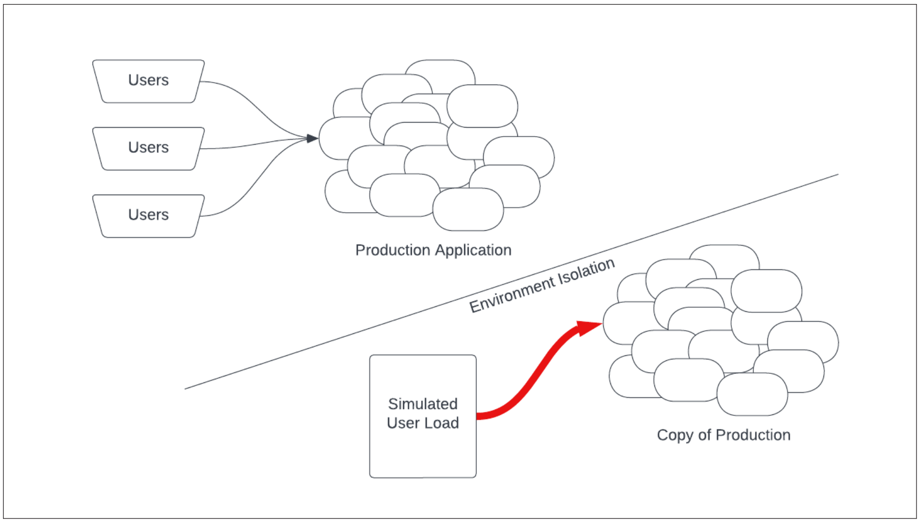

Figure 3-9. Load testing using simulated users on a copy of production

Here you see a production application with real users in the top left hand section. Then, in the lower right hand section is an exact copy of the production setup to work with a simulated user load. This environment allows you to test various user loading scenarios on an equivalent of a production environment in a very accurate manner. This can be done independently and completely isolated from the existing production environment, yet in a manner that is virtually identical to the production environment.

Adaptive architectures allow setting up many different testing scenarios such as load testing in production equivalent environments without impacting existing production environments.

Cost of adaptive architecture in increased complexity

Adaptive architectures are, on the surface, more complex than non-adaptive architectures. The increased flexibility and agility gives you the ability to be more flexible and agile. The increased ability to programmatically modify your infrastructure means you can modify your infrastructure to be as complex as you want to. Unless managed appropriately, flexibility breeds complexity.

As we learned in chapter 1, increased complexity increases technical debt and decreases the reliability and availability of your application. Increased complexity leads to fragility (instability of the application) or rigidity (resistance to making changes to the application). So, unless properly managed, adaptive architectures can lead to increased fragility or increased rigidity, neither is good.

Adaptive architectures provide you all of the great advantages discussed previously, yet they also can lead to a rigid and/or fragile organization due to increased complexity. So, how do you avoid complexity while leveraging the benefits of an adaptive architecture?

Interestingly, the way you reduce complexity in adaptive architectures is to use the same techniques you use to reduce general software complexity in large software applications.

This means leveraging common best practices that help reduce the cognitive load of large application systems. Best practices such as:

Modularization

Breaking software into smaller modules, such as services and microservices, is a great way to compartmentalize complexity and hence reduce the amount of complexity you need to keep in your mind at any given point in time. Similar techniques work to compartmentalize complexity in adaptive architectures. Modularizing your infrastructure and managing the modules individually, reduces the impact of complexity in your application infrastructure.

For example, when architecting the front-end load balancer for an application, you treat the application architecture itself like a black box. Similarly, when you focus on creating a dynamic infrastructure for an application using multiple services, you can treat the database high availability strategy you require as a black box, ignoring the details until later.

Loose Coupling

Reducing tight dependencies between software services creates separation that reduces friction in large scale application development. Similarly, in adaptive architectures, reducing the coupling between architectural modules reduces the complexity in working with adaptive architectures. By reducing the required interaction between infrastructure components, you reduce the complexity of those components.

For example, the interaction between a front-end request cache and an application service should be restricted to basic HTTP caching protocols. There should be no deeper integration between the cache and the application than those basic HTTP protocols. While it might seem wise to build a backdoor cache invalidation system connecting the application to the cache, avoid that in lieu of basic HTTP cache invalidation commands already in the standard HTTP protocol.

Reuse components

Reusing software components is a classic model to reduce software system complexity, and the same technique works with adaptive architectures. By using common, reusable infrastructure modules repeatedly in different places in your architecture, you can reduce complexity by increasing modularity and loose coupling.

For example, if every server you deploy has the same basic structure and components, you'll have fewer variations to worry about. If each service is structured identically (or using one of a few different infrastructure structures), you can reuse the same patterns in multiple services, reducing complexity. Using one or two standard inter-service communications models reduces the complexity involved in each service picking their own communications protocol. Allowing individual services to control the software installed and the directory structure of the underlying servers may ease some tasks for those services, but it increases the number of moving parts for the architecture as a whole.

Standardize

Restricting choices by standardizing on specific components reduces complexity, often improving agility and time to value.

For example, AWS has hundreds of different variations of sizes and shapes of servers to choose from when creating an adaptive architecture. Any of those hundreds of variations can be used in the same application. However, by reducing the allowed list of choices to a select few that covers the variation needs of your application, you now only have to deal with a few variations, rather than the full set of hundreds. This reduces complexity substantially.

Besides leveraging best practices such as these, there are other strategies that can reduce your architectural complexity. Consistency, repeatability, regularity, all of these things help to reduce complexity.

EXAMPLE: DECREASING COMPLEXITY WITH TIERED SECURITY

Let's take a look at an example of managing and reducing complexity in an adaptive architecture. In this example, we're going to look at a cloud-based security model and how you can use some of the above techniques to reduce the complexity in the security setup.

Take a look at Figure 3-10.

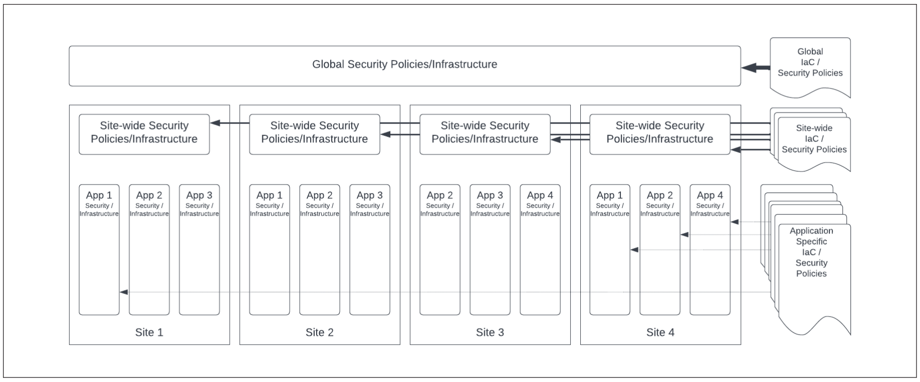
Global Security Policies/Infrastructure
Global IaC / Security Policies
Site-wide Security Policies/Infrastructure
Site-wide IaC / Security Policies
App 1 Security / Infrastructure
App 2 Security / Infrastructure
App 3 Security / Infrastructure
App 4 Security / Infrastructure
Application Specific IaC / Security Policies
Site 1
Site 2
Site 3
Site 4

Figure 3-10.

This diagram shows a simplified security model for an enterprise. This enterprise has four applications that are spread across four different geographic sites. Site 1 and Site 2 hosts applications 1,2, and 3. Site 3 hosts applications 2, 3, and 4. Site 4 hosts applications 1, 2, and 4.

For security, there are security policies and security infrastructure requirements for every app segment on every site, plus each site has security requirements. This means there are 16 different and distinct dynamically changing security requirements that must be managed. If they are each managed independently, keeping track of all of these requirements will be nearly impossible, especially as the number of servers, sites, and applications grow. If this was a reasonable size enterprise, there might be tens of thousands of distinct security and infrastructure requirements for each component in the system.

Obviously, it would be best if all the policies and infrastructure were the same, so that there is only a single policy to track. Realistically, however, this isn't going to be the case. Different locales have different security requirements. Different applications have different security requirements. The complexity grows exponentially as the number of sites and applications grows.

The solution is to implement a tiered, or hierarchical security model, using the modular, loose-coupling, and reuse practices described earlier. In this model, there is a set of standard, global, security policies and infrastructure requirements, that apply to all applications in all locales. Next, there is a set of site-wide security policies and infrastructure requirements that are set and established at the site/ locale level. Finally, there are a set of application specific policies that apply to each application at the third level.

All general policies and standardized infrastructure requirements are included at the top, global, policy level. These apply to everything. The site-wide/ locale policies only contain the policy exceptions and how they deviate from the global policy. Then, at the application specific level, the policy exceptions that apply at a specific application are applied here.

The desire is to put as many policy and requirements into the global security policy as possible. Everything that applies company-wide should be specified once at the global level. These policies and requirements are applied universally across all sites and all applications. This would include things like user password requirements, security rules to keep BOTS at bay, edge firewall requirements, etc.

The site-wide policies should only contain exceptions to these global policies that apply at the given locale. For example, locales in EU that must follow EU's GDPR requirements would have these exceptions described in the EU locale policies, or they might have to deal with policies required for AWS regions that are different for Azure regions. The end result is a set of policies that are, mostly, described and implemented once (globally), but can be adjusted and modified as needed for a given locale. These site-wide policies can depend on yet ignore the rules specified in the global policy. The site-wide policies, for examples, don't have to deal with BOTS, because the global tier is dealing with them.

Finally, application specific exceptions are described at the application-specific policy level. These policies might include requirements such as which network ports that application requires, what type of traffic is expected, and scaling requirements. This layer can ignore those policies established by the locale based policies and the global policy, and only focus on the ones needed that are unique to this application.

As many policy and infrastructure requirements as possible should be specified at the global level, with fewer and fewer requirements as you move down the tiers to the more finer grain, application level requirements, which should specify as few requirements as possible.

The net result is rather than tens of thousands of distinct security policies, you have three tiers, with only the required detailed exceptions at the lower, finer granular level. The vast majority of the requirements are specified at the single, global level.

This sort of model puts process and procedure around the complexity of managing a completely agile security model. It makes use of *modularization* with the tiers of control, *loose coupling*, since the layers are independent of each other as much as possible, *reused components* to reduce the variations, and overall a *standard model* that impacts how and where changes can be made.

There is nothing magic about this tiered model, but models such as this can dramatically simplify the operating and development models of your application that is employing adaptive architecture techniques.

Often policies in a model like this are handled in an out-of-band management tier that manages the code, configuration management, infrastructure and policy rules, and analytics in a general fashion, keeping those requirements global yet safely isolated from the three production tiers. The management *control plane* tier is independent of the other three tiers and provides the management and control for all the other tiers.

Summary

Adaptive architectures are a fantastic tool to improve the creation and operation of large modern applications. However, used inappropriately, adaptive architectures can increase your application complexity, leading to fragility and rigidity in thought, process, and design as an organization. Using the same best practices that have been used reliably in building large scale software applications, we can leverage adaptive architectures yet manage the increased complexity to allow us to gain the advantages of the adaptive architectures without the burden of increased complexity.

All of this means more agility, which means you can build more secure applications while enabling competitive business innovation.

About the Author

Lee Atchison is a recognized industry thought leader in cloud computing, and the author of the best selling book *Architecting for Scale*, published by O'Reilly Media, currently in its second edition. Lee has 34 years of industry experience, including eight years at New Relic and sever years at Amazon.com and AWS, where 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 to a new service-based architecture. Lee has consulted with leading organizations on how to modernize their application architectures and transform their organizations at scale. Lee is an industry expert and is widely quoted in publications such as *InfoWorld*, *Diginomica*, *IT Brief*, *Programmable Web*, *CIO Review* and *DZone*. He has been a featured speaker at events across the glove from London to Sydney, Tokyo to Paris, and all over North America. LinkedIn profile: https://www.linkedin.com/in/leeatchison.