**17**

**Rearchitecting Legacy Systems**

Today's organizations are operating in a challenging environment. The pace of change is unprecedented. Regulators and institutions are imposing new reporting and security requirements, new technologies are disrupting consumers' expectations and perceptions, and the ecosystem is constantly evolving as new players enter the market. As a result, organizations are redefining their business models to provide the customer focus, agility, and technology they need to attract talent, be competitive, and grow.

Application modernization has become a critical component of these new business models to rapidly stand-up dev/test environments, experiment with new ideas, and then develop new products and services. In addition to eliminating the need to invest in expensive and cumbersome infrastructure, the new system enables innovation through the broad set of technologies it makes available.

Legacy systems are the applications that have been deployed in your data center for decades without undergoing many changes. In a fast-changing technology environment, these systems get outdated and are challenging to maintain. Legacy systems are defined by their age and their inability to meet growing business needs due to the underlying architecture and technology.

Often, large enterprises deal with legacy applications to run crucial day-to-day business tasks. These legacy systems are spread across industries such as healthcare, finance, transportation, manufacturing, and supply chain industries. Companies have to spend heavily on the maintenance and support of these systems, which warrants the need to architect legacy systems.

Rearchitecting and modernizing legacy applications helps organizations be more agile and innovative and optimizes cost and performance.

In this chapter, you will learn about challenges and issues with legacy applications and techniques to rearchitect them. Rewriting complex legacy applications may pose an additional risk of business disruption, so you will learn about refactoring applications or considering the option to migrate into a more flexible infrastructure. The following topics will be covered in this chapter:

* Learning the challenges of legacy systems
* Defining a strategy for system modernization
* Looking at legacy system modernization techniques
* Defining a cloud migration strategy for legacy systems
* Mainframe migration with the public cloud

By the end of the chapter, you will have learned about various challenges and modernization drivers for legacy systems. You will learn various strategies and techniques for the modernization of legacy systems. As the public cloud is becoming a go-to strategy for many organizations, you will also learn about the cloud migration of legacy systems.

**Learning the challenges of legacy systems**

A legacy application presents significant challenges for an organization. On the one hand, there are critical applications that an organization has been using for decades. On the other hand, legacy applications are holding back the organization's pace of innovation.

In a hugely competitive environment, end users are looking for the most modern, technologically advanced applications. All new features usually come with the latest software, and legacy applications limit your ability to add those features and benefit end users. The following diagram shows some significant challenges that organizations are facing with legacy systems:

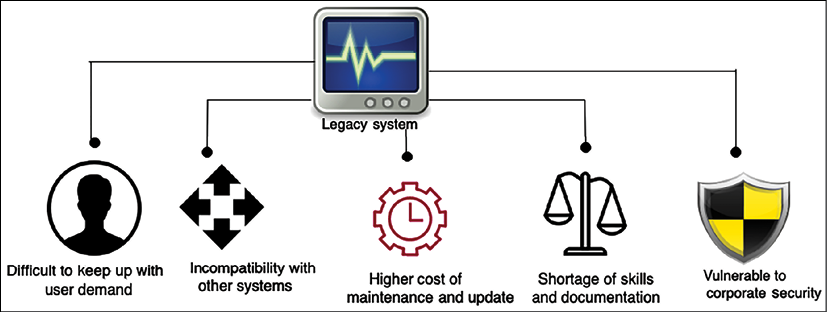


Figure 17.1: Challenges with a legacy system

At the top level, the CIO owns an application portfolio that represents the business processes of the organization. The portfolio of applications can be from hundreds to thousands depending upon the organizations' size. There are four discernable decision patterns in a large enterprise:

* **Chief Information Officer** (**CIO**): The CIO owns the business applications decisions. Here, the decision is developed in-house to modernize an old app or prioritize the use of SaaS. Common priorities are consumer experience/user experience and supply chain.
* **Chief Security Officer** (**CSO**): As the CIO makes application-priority decisions, the transformation or modernization of those apps drives the modernization of the security model. This means a complete revisit of old on-premises approaches to security, a move away from hardware appliances, a move to the cloud, and working with third-party software vendors who have approaches for security for a cloud world.
* **VP of Big Data**: They receive data streams like never before. The possibilities are so much more flexible and scalable that many companies make big data their second driver for the use of the public cloud, after developer productivity.
* **VP of Infrastructure and Operations**: They have to go through substantial process and people skill changes, as this team has to adopt new tools, abandon old tools, and move to a mixed world of ITSM and DevSecOps for new modernized apps using new tools across serverless, containers, and SaaS. This team is modernizing their software tooling at a prolific pace but is also under pressure to reduce costs while being more agile.

Before we dive into the solution, it's important to understand the issues clearly. Let's explore the challenges of legacy systems in more depth to understand them better.

**Difficulty in keeping up with user demand**

Customer focus is the key to business success and being unable to keep up with the latest technology trends can harm a business significantly. You can take the example of Nokia, which used to lead the global mobile phone market. As smartphones came into play nearly a decade ago, Nokia still stuck with a legacy system, resulting in near bankruptcy. It was a similar story with Kodak—one of the largest businesses in the camera industry. Kodak was unable to move with digital innovation and adopt it into its systems, which resulted in Kodak becoming bankrupt in 2012. There are many such examples of large enterprises being unable to survive due to a lack of legacy modernization and innovation.

In the current climate of fast-changing technology and fierce competition, users are very demanding. Now, organizations have to change as per the user's terms, as they have multiple choices. As technology moves, the user moves with it and starts using the most recent and popular applications. Your competitors can jump ahead if they are providing new features that the user needs.

A legacy system also poses challenges for enterprise applications with an internal user base. An old system built on mainframes primarily uses the command line, which is not user-friendly in the digital age. In contrast, new-generation workers demand a more user-friendly approach to perform their routine tasks. However, you may face significant resistance from management, who may have been working with legacy systems for decades and are used to them.

The technology at the core of large enterprises is often outdated and comprises systems dating back decades. Organizations running their core systems on legacy, on-premises technology face severe challenges when it comes to enabling modern experiences for their customers. Many systems are the product of multiple mergers and acquisitions, resulting in fractured data siloes, excessive infrastructure costs, and slow development time. This creates inefficient processing and decision making, a lack of business agility, poor customer responsiveness, and high maintenance costs. Under these conditions, it's challenging for IT to meet the modern needs of internal stakeholders and customers.

**Higher cost of maintenance and update**

As legacy systems have been all set up and working for decades, they may look less expensive. But over time, the total cost of ownership turns out to be higher, as support and updates for old systems are usually more costly.

Often, those updates are not available out of the box, and lots of manual workarounds are required to maintain the system. Most legacy systems are not very automation-friendly, resulting in more human effort.

Legacy systems mostly have a large chunk of proprietary software, which results in significantly higher license fees. In addition to that, old software no longer receives support from providers, and buying additional support out of the life cycle could be very costly. On the other hand, modern systems mainly adopt open source technologies that drive the cost down. The operational outage from a legacy system can take more time and drive up operating expenses. People with the skill set to maintain legacy systems (such as DB2, COBOL, Fortran, Delphi, and Perl) are hard to find, increasing hiring costs and system risk significantly.

Legacy systems are pretty significant in the aspect of code. Unused code adds another layer of unnecessary maintenance and complexity to a system. Legacy applications have been in operation for decades and, over time, many changes will have been accommodated without code being cleaned, which amounts to lots of technical debt. Any initiative to reduce technical debt could be risky due to unknown impacts and dependencies. As a result, organizations are forced to invest in unnecessary code and system maintenance for fear of breaking the system by making any significant changes.

However, modernizing legacy systems may be costly due to unknown dependencies and outages. A careful **cost-benefit analysis** (**CBA**) needs to be considered and the **return on investment** (**ROI**) needs to be determined when deciding to proceed with modernization. As stakeholders don't see the immediate benefit of modernization, procuring finances for legacy modernization can be challenging.

**Shortage of skills and documentation**

Legacy technologies (such as mainframes) have multiple complex components that depend on each other. They are extensive proprietary and costly servers that are not readily available if someone wants to develop skills on their own. It is challenging to retain application development resources and even more challenging to hire people with hands-on experience with old technology and operating systems.

Often, legacy systems are two or more decades old, and most of the workforce with the relevant skills to manage them has retired. Also, these systems may not have the documentation to keep a record of the years of work that went into them. There are chances of significant knowledge loss as an old workforce rotates with a new workforce. A lack of knowledge makes it risky to change the system due to unknown dependencies. Any small feature requests are challenging to accommodate due to system complexity and skills shortages.

New cutting-edge technologies such as big data, machine learning, and the**Internet of Things** (**IoT**) are built on new technology platforms. As new technologies are not well integrated with legacy systems, an organization may lose out to a competitor if it cannot use the full capabilities of emerging technologies. A modern system helps build an organization's brand as an innovative company where most of the new generation of the workforce wants to work. Development and training are an even more significant source of expense for legacy technologies.

Often, automation helps to reduce costs by reducing human effort. Many tools are available in modern systems to build automation—such as DevOps pipelines, code review, and automation testing—that a legacy system may not utilize, resulting in additional cost.

**Vulnerable to corporate security issues**

Security is the top priority for any organization and system. A legacy application that runs on an old operating system (such as Windows XP or Windows 2008) is vulnerable to security issues due to a lack of vendor support. Software vendors continuously determine new security threats and release patches to accommodate them in the latest software version to secure them. Any legacy software announced as **End of Life** (**EOL**) from a vendor doesn't get a new security patch, which leaves your application running in the old software version, exposed to several security threats.

System health checks are often ignored for legacy applications, which makes them more vulnerable to security attacks. The skills gap makes it difficult to provide continuous support and help, which means systems are run in an insecure manner. A single vulnerability can pose a high risk of exposing your application, database, and critical information to attackers.

In addition to a security vulnerability, legacy applications are hard to maintain due to compliance. As compliances keep changing over time to enforce more tight security around data handling and usage, legacy systems require changes to adhere to local governance and compliance needs.

For example, the European Union's new **General Data Protection Regulation** (**GDPR**) compliance requires each system to enable users to request to delete their data. While modern systems can provide these features out of the box in an automated and self-service manner, this may need to be performed manually and becomes more complex in legacy systems.

Adhering to compliance needs can lead to more operation costs and time-consuming maintenance.

**Incompatibility with other systems**

In addition to end users, every system needs to integrate with other IT systems. Those systems may be associated with different departments, clients, partners, or suppliers. The various systems need to exchange data in a standard format that evolves. Almost every few years, files and data format standards are changed to increase data exchange efficiency, and most systems need to change to adopt them. Hard-to-change legacy systems that stick to using an old format could result in system incompatibility and a system that your suppliers and partners may not want to use. An inability to accommodate standard needs adds significant risk to businesses due to complex workarounds and lost productivity.

Adding a workaround for simple business needs may make a system more complex. Modern systems are built on a service-oriented architecture, making it easier to accommodate any new requirement by adding a new service independently. Old systems are often built with a monolithic architecture, and adding any new feature means you need to rebuild and test the entire system.

Modern architectures are API-oriented and can be easily integrated with other systems to offload heavy lifting. For example, a taxi-booking app uses Google Maps for **Global Positioning System** (**GPS**) navigation or Facebook and Twitter for user authentication. A lack of APIs makes these integrations harder in a legacy system, resulting in complex custom code.

As load increases from another dependent upstream system, a legacy application can face a *scalability issue*. Often, legacy applications are built with a monolithic architecture and are hardware-dependent. Scalability is a big challenge with a monolithic system, as it cannot scale horizontally due to hardware dependency and vertical scaling being limited to the maximum system capacity. Breaking monolithic applications into microservices can help with scaling challenges and help keep up with the load.

In addition to software maintenance, legacy applications are also costly for hardware infrastructure as they run on a particular version. They spread across multiple databases with duplicate data and similar functionality. Due to their monolithic nature, it's hard to consolidate and use the flexibility of cloud-based infrastructures to save costs. Let's look at some critical advantages of system modernization.

**Benefits of system modernization**

Creating a future digital strategy by addressing the growing need for legacy system modernization can have many advantages, as shown in the following diagram:

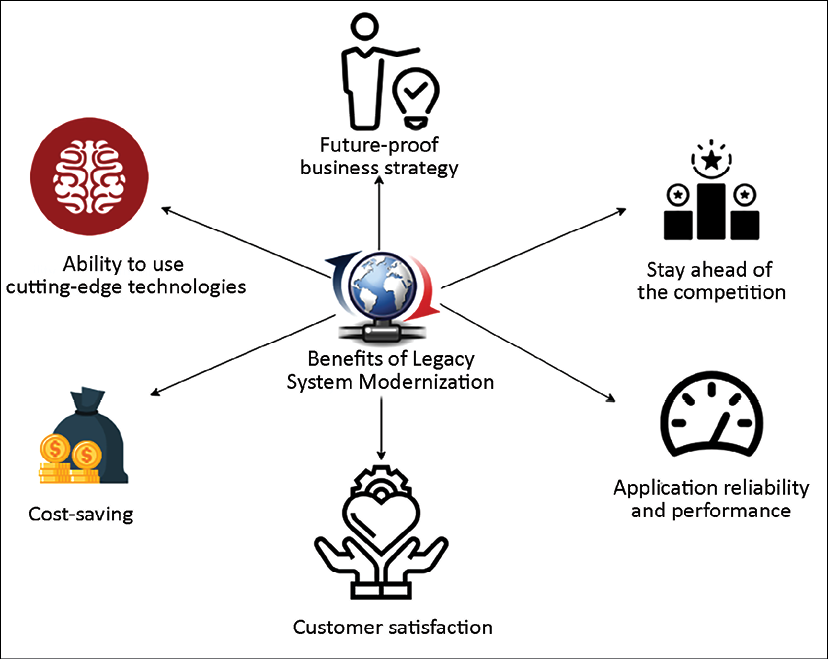


Figure 17.2: Benefits of legacy system modernization

The following are the significant benefits of application modernization:

* **Customer satisfaction**: Using the latest technology gives a better **user interface** (**UI**), **user experience** (**UX**), and an omnichannel experience. Consumers have grown accustomed to real-time access to information from any device, any location, at any time through their personal experiences. You don't need to build different variations of the UI; it can be built once and deployed across devices such as laptops, tablets, and smartphones. A fast and slick UI leads to better customer experience and business growth.
* **Future-proof business strategy**: Modernizing your application allows you to be more agile and innovative. The team can accommodate the changing needs of the business comfortably and evolve with new technology.
* **Stay ahead of the competition**: Users are always looking for the latest stuff and tend to move to new applications that give a better experience. The modernization of your application helps you to stay ahead of the competition by following the latest trends. For example, voice integration is being widely provided in apps, and you can enhance security with face detection. This is only possible when your application is adopting the latest technology.
* **Application reliability and performance**: Every new version of a software API and an operating system tries to address and improve performance issues. Using the latest software and hardware helps you to achieve better performance, scalability, and high availability. Application modernization helps you to reduce operational outages and enhance security.
* **Ability to use cutting-edge technologies**: Legacy systems prevent you from getting insights from data that could help you grow your business. By modernizing your database and creating a data lake, you can use big data and machine learning to get all kinds of insights. This also helps you to retain employees when people get the opportunity to work with new technologies.
* **Cost savings**: Overall, any modernization leads to cost savings by reducing operational maintenance and providing a more natural upgrade. Utilization of open source software reduces licensing costs, hardware flexibility helps to adopt a cloud pay-as-you-go model, and automation reduces the human resources needed for routine jobs and improves overall efficiency.

By migrating legacy core systems, organizations can modernize their core systems to reduce the cost of ownership, automate manual back-office processes, eliminate data silos, improve customer experience, and launch new market-facing applications faster.

However, there are several benefits of legacy system modernization, but they can be very complex and require lots of effort. A careful assessment needs to be conducted to take the right approach. Let's explore the assessment techniques of a legacy application.

**Defining a strategy for system modernization**

Often, a legacy system gets left out of an enterprise's overall digital strategy, and issues get addressed on an as-needed basis. Taking a reactive approach holds back organizations from executing overall system modernization and benefits.

If your legacy system has serious business challenges, such as security and compliance issues, or cannot address the business need, you can take a **big-bang approach**. In the big-bang method, you build a new system from scratch and shut down the old system. This approach is risky but addresses a business need that can be mitigated from the existing legacy system.

The other approach you can take is a **phased approach**, where you upgrade one module at a time and keep running both the old and the new systems. A phasedapproach is less risky but takes a long time and may be more expensive as you need to maintain both environments, with the concomitant increased network and infrastructure bandwidth.

Approaching your application portfolio, prioritizing certain applications, and having an overall plan is the first step. As you use the cloud, you design a new operating model, and you end up with a combination of tools. You can choose to use third-party tools to frame your needs and your tool preferences. And finally, you can use a consulting partner for a more successful and faster time to complete migration and modernization projects.

Taking any of these approaches can provide various benefits once the modernization of an application is completed.

**Assessment of a legacy application**

There may be multiple legacy systems in an organization, with tens of thousands to millions of code lines. In a modernization situation, a legacy system needs to align with the business strategy and the investment cost. Also, it is possible to reutilize some parts of it or completely write it from scratch, but the first step is to conduct the assessment and better understand the overall system. In the Assessment phase, the solution architect needs to make it easy to assess more quickly and make informed decisions. Assessments can be done in days and weeks. The following points are the primary areas that solution architects need to focus on when conducting an assessment:

* **Technology assessment**: As a solution architect, you need to understand the technology stack used by the existing system. If the current technology is entirely outdated and lacks vendor support, you might need to replace it entirely. In the case of a better version of the technology being available, then you can consider upgrading. Often, newer versions are backward-compatible, with minimal changes required.
* **Architecture assessment**: You need to understand the overall architecture to make it future-proof. There may be a case where you need to make a minor upgrade to the technology, but the overall architecture is monolithic and not scalable. You should audit the architecture in the aspects of scalability, availability, performance, and security. You may find significant architecture changes are required to align the application with business needs.
* **Code and dependency assessment**: Legacy systems often have hundreds of thousands of lines of code in a monolithic setting. Various modules tied to each other make the system very complicated. Code that appears not to be used in one module might impact other modules if it is removed without due diligence. These code lines may have been written decades back and may have missed regular refactoring and review. Even if the technology and architecture look fine, you need to determine if the code is upgradable and maintainable. We also need to understand if any UI-related upgrades are required to make the user experience better.

As a solution architect, you want to determine the dependencies across various modules and code files. Modules may be tightly coupled, and you need to define an approach to perform simultaneous upgrades when modernizing the overall architecture. During your assessment, you may find the following patterns:

* First, many customers realize that they have many old apps that do not relate well to the future business model; they can be retired. For example, around 10-20% of the app portfolio can be retired.
* Second, thousands of SaaS vendors didn't exist 5-7 years ago; these SaaS vendors can replace many on-premises apps. For example, most large customers have landed on Salesforce as a CRM platform. This shift to SaaS shrinks down the operational portfolio managed by IT operations—it still presents security and identity work but has lower operations costs.

Then, decisions can be made to lift and shift and, during a move, re-platform the operating system, database, or language to reduce costs, such as customers choosing to re-platform from Windows Server to Linux and from Oracle to Postgres to reduce database licensing costs. If you choose to modernize, there should be a focus on modernizing apps that are true differentiators to your business. Let's look into the modernization approach.

**Defining the modernization approach**

For stakeholders, there may be no immediate incentive for application modernization. You need to choose the most cost-effective method and deliver results faster. The following diagram shows the modernization approach:

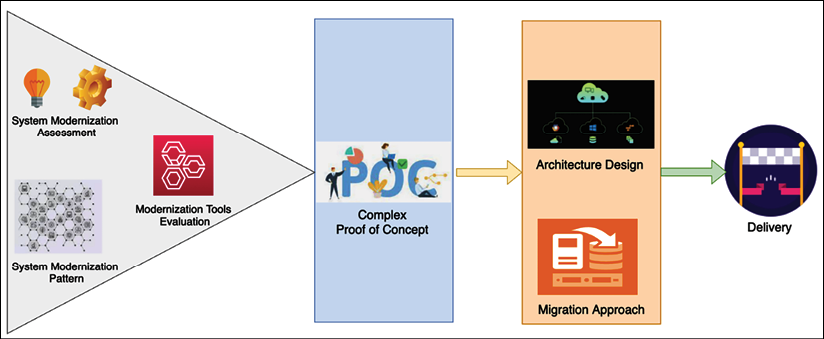


Figure 17.3: The legacy system modernization approach

After your system assessment, you need to understand the existing architecture pattern and its limitations. As per your tech stack, you need to evaluate migration tools. For example, you may choose to use an emulator for mainframe migration or vCenter if you're rehosting your application to VMware. You can select various modernization approaches and create a **proof of concept** (**POC**) to identify gaps. Some approaches are listed here:

* **Architecture-driven modernization**: The architecture-driven approach is required to achieve the most agility. Often, an architectural approach is language-independent and platform-independent by applying service-oriented patterns, which gives the development team the flexibility to be more innovative. You may want to choose this approach if your assessment shows that you need to make significant architectural changes. Start implementing the most critical feature first and then build a POC to highlight the gaps and the effort required. Take the microservices approach to achieve scalability and ensure better integration with other systems, depending on your legacy application.
* **System re-engineering**: In the re-engineering approach, the solution architect needs to understand the legacy system in depth and perform reverse engineering to build a new modernized application. You need to be sure to make technology choices that help you to create a future-proof system. You may want to take this approach if the legacy system is over-complicated and requires long-term projects. Start with application modernization first and upgrade the database as a final cutover in a phased approach. You need to build a mechanism where the legacy and upgraded modules co-exist, with the ability to communicate in a hybrid manner.
* **Migration and enhancements**: You can use migration and minor enhancement approaches if your existing system technology works relatively well but is restricted due to hardware limitations and cost. For example, you can lift and shift the entire workload to the cloud for better infrastructure availability and cost optimization. In addition to that, a cloud provider extends several out-of-the-box tools, which help you to make changes more frequently and apply better automation. A migration approach enables you to modernize your application with less effort and makes it future-proof, keeping it relevant for the long term. However, lift and shift is limited and may not be suitable for all kinds of workloads.

As you aim to migrate and modernize, make sure to consider specific IT domains that require substantial redesign and modernization. This modernization includes developer operating system environments as it affects patch management. Security, network, and identity are next and offer a great opportunity for scalability, resilience, and cost reduction. After that comes storage, backup, and database tools as more apps move to the cloud. Also, you need to modernize your monitoring and management tools, and all these require training and re-skilling. Let's look into various strategies for modernizing legacy systems.

**Looking at legacy system modernization techniques**

As per your existing application analysis, you can take various approaches to upgrade your legacy system. The most straightforward approach is migration and rehosting, where you don't need to change the existing system. However, a simple migration may not solve the long-term problem or provide any benefits.

You can take a more complex approach, such as rearchitecting or redesigning the entire application if the system no longer meets business needs. The following diagram illustrates the impact of the various methods:

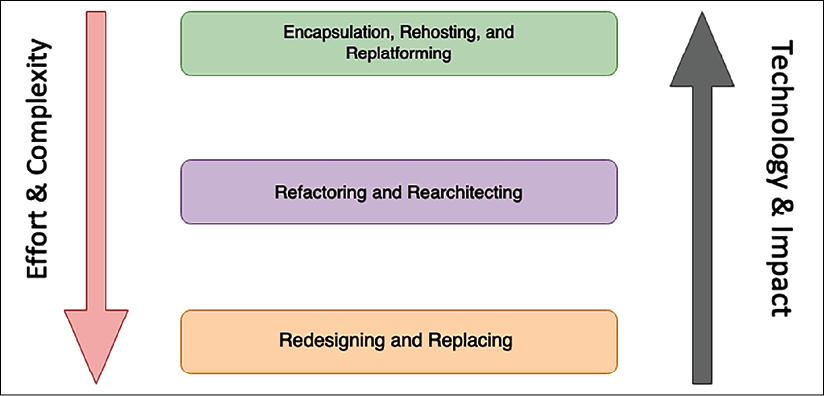


Figure 17.4: Legacy system modernization techniques

Let's look in more detail at the various modernization techniques shown in the preceding diagram.

**Encapsulation, rehosting, and replatforming**

**Encapsulation** is the most straightforward approach. If the system is business-critical and needs to communicate with other applications running on the latest technology, you may want to use this approach. With encapsulation, you need to build an API wrapper around your legacy system, which will allow other business applications to communicate with a legacy application. An API wrapper is a common approach whereby you start migrating your applications to the cloud but keep the legacy application in the on-premises data center for modernization in the later phase. You can choose the encapsulation option if your legacy code is well written and maintained but, again, you will not be able to benefit from technology advancements and hardware flexibility.

The **rehosting** approach is also among the most straightforward strategies, whereby you migrate your application into another hardware provider, such as the AWS cloud, without any code changes. Again, as with encapsulation, the rehosting option can reduce costs due to vendor contracts but you may not benefit from technology advancements and hardware flexibility.

An organization often takes this approach when it needs to move out of an existing contract quickly. For example, you can take the first step to the cloud in phase one and apply modernization in phase two.

The **replatforming** approach may get a bit more complex than the rehosting approach but will provide immediate benefits. Organizations often choose this approach if the server reaches **End of Life**(**EOL**), where no support is available, and an upgrade is necessary to handle security issues. For example, if Windows Server 2008 is reaching EOL, you may want to upgrade the operating system to the Windows Server 2019 or 2022 version. You need to rebuild your binaries with the new operating system and carry out testing to make sure everything works properly, but there are no significant code changes. Again, as with rehosting, with replatforming you may not benefit from technology advancements. However, it will allow you to have continuous support from the vendor.

While the preceding three approaches are the simplest ones, they cannot provide the full benefit of the application upgrade. Let's look at approaches that help you to take full advantage of application modernization.

**Refactoring and rearchitecting**

In the **refactoring** approach, you can refactor your code to accommodate the new system. In refactoring, the overall architecture will be the same, but you are upgrading your code to suit the latest programming language and operating system version. You can refactor the portion of code to apply automation and perform feature enhancement. If your technology is still relevant and can accommodate business needs with code changes, you may want to take this approach.

In the **rearchitecting** approach, you decide to change the system architecture by reutilizing the existing code as much as possible. For example, you may create a microservices architecture out of your existing monolithic architecture. You can take one module at a time and convert it into a service-oriented architecture by giving each module a RESTful endpoint. The rearchitecting option helps you achieve the desired scalability and reliability; however, overall performance results may be average due to the utilization of existing code.

**Redesigning and replacing**

The **redesigning** approach is the most complex but provides the maximum benefit. You can choose this approach if the legacy system is completely outdated and cannot accommodate business needs at all. With redesigning, you need to build the entire system from scratch while keeping the overall scope intact.

The following diagram shows the legacy mainframe system migration to the AWS cloud:

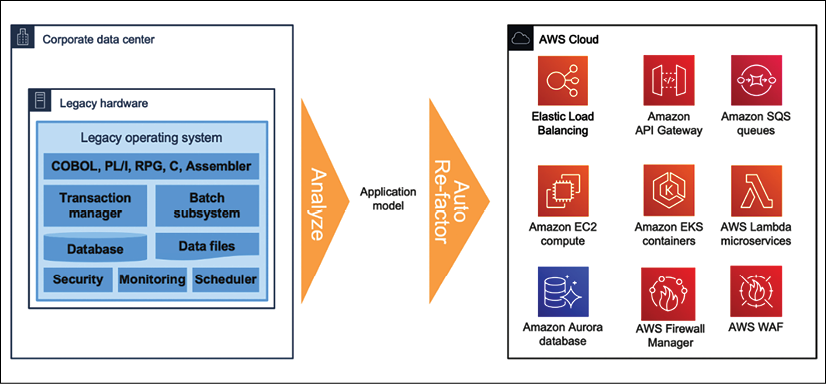


Figure 17.5: Legacy mainframe system modernization to the cloud

Here, a legacy mainframe system is rearchitected and refactored to similar cloud services as a *modernization approach*. Building a cloud-native application helps you utilize and fully benefit from cloud services in aspects of scalability, performance, reliability, and cost. It helps your team to be more agile and innovative by accommodating rapidly changing technology in your system.

Redesigning a legacy system requires a long-term project with lots of effort and increased cost. Before kicking off modernization, as a solution architect, you should do careful analysis if any SaaS product or **commercially available off-the-shelf** (**COTS**) products can handle your business need with a lower cost. It is essential to do a **Cost Benefit Analysis**(**CBA**) between *redesign* and *purchase* before proceeding with the redesigning option.

Sometimes, it's more beneficial to replace the existing legacy system with new third-party software. For example, your organization may have a decade-old **Customer Relationship Management** (**CRM**) system that cannot scale and provide the desired feature. You can look for the option to subscribe to SaaS products such as Salesforce CRM to replace the legacy system. SaaS products are subscription based and offer per-user licenses, so they may be the right choice if you don't have many users. For a vast enterprise with thousands of users, it may be more cost-effective to build its application. You should conduct a CBA to understand the ROI when investing in SaaS products.

**Defining a cloud migration strategy for legacy systems**

As the cloud becomes ever more popular, more organizations are looking to migrate into the cloud for their legacy application modernization needs. You learned about various cloud migration techniques in *Chapter 5*, *Cloud Migration and Hybrid Cloud Architecture Design*. The cloud provides you with the flexibility to scale your application while keeping costs low and helps you to achieve desirable performance, high availability, and reliability while maintaining application security.

Cloud providers such as **AWS**, **Microsoft Azure**, and **GCP** provide many options out of the box that can help you to modernize your system. For example, you can take a serverless approach to build a microservice using AWS Lambda and Amazon API Gateway, with Amazon DynamoDB as the backend. We discussed various legacy system modernization techniques and their application in the context of moving to the cloud in the previous section. The flow illustrated in the following diagram will help you decide whether to use cloud migration to modernize your legacy application:

Diagram

Description automatically generated

Figure 17.6: Cloud migration path for legacy system modernization

As shown in the preceding diagram, if your application is still heavily used by businesses and generates revenue, you may want to continue with minimal changes. In that situation, you can refactor your application into the cloud or re-platform it into the cloud if the server is reaching EOL.

If you don't want to change existing applications to sustain business and still want to move to the cloud entirely to save and optimize costs, then take the *lift and shift approach* to rehost the legacy application in the cloud. If your legacy application is replaceable, then you can buy a cloud-native SaaS version of the product and retire your legacy application. Sometimes, you may want to retain your legacy system in the on-premises data center if there are too many business dependencies and it cannot move into the cloud due to incompatibility.

You should perform a **total cost of ownership** (**TCO**) analysis to understand the advantages of moving to the cloud. It is recommended to take the most complex module of the legacy application and build a POC to ensure your entire system will be cloud-compatible before starting the full project. A detailed POC covering the critical business cases will help you identify gaps and reduce the migration risk significantly.

**Documentation and support**

For the long-term sustainability of a new system and graceful migration to it, make sure to prepare proper documentation and support. Provide documentation for your coding standards that everyone can follow and that helps to keep the new system up to date. Keep your architecture documents as working artifacts and keep them updated as technology trends change. Keeping your system updated will ensure that you don't fall into the legacy system modernization situation again.

Prepare a comprehensive runbook to support new and old systems. You may want to keep the old system for some time until the new system can accommodate all business requirements and run satisfactorily. Update the support runbook and ensure that you don't lose knowledge due to employee attrition and that the overall knowledge base is not processed in a people-dependent manner.

Keeping track of system dependencies helps you to determine the impact of any changes in the future. You will learn more about documentation in the next chapter.

Prepare training content to train staff on the new system and ensure they can support it in case there's an operational outage.

**Mainframe migration with the public cloud**

Many enterprises are moving their mainframe workloads to the cloud to take advantage of factors such as cost reduction, increased agility, technical debt reduction, digital strategy support, the legacy mainframe skills gap, and data analytics. Mainframe workloads are more challenging to migrate than x86-based workloads because legacy mainframe applications are often developed and deployed in a tightly coupled manner. For example, a mainframe application might include programs that are used by a number of subsystems or are directly called by other applications. In these cases, changes made to the underlying programs also affect the associated subsystems and applications.

For legacy applications, you need to take an incremental approach, where the migration is planned in waves, as a best practice. This approach helps to reduce risks because you select and prioritize closely related applications to be migrated together. However, this approach sometimes isn't as straightforward for mainframe migrations because the mainframe application code can use temporal coupling (invoked synchronously) or deployment coupling (using linked modules). Migrating the coupled application code affects dependent applications and therefore carries some risks. To reduce these risks, you can decouple the mainframe application code without impacting dependent applications. From a code migration perspective, the two main types of legacy mainframe applications are standalone applications and applications with shared code. Let's look into the details of each migration pattern.

**Migrating standalone applications**

Let's assume that there are two applications, A and B, that are standalone mainframe applications. Each application consists of programs and subprograms that it uses exclusively.

Because the applications are self-contained, you can group the **COBOL** programs and subprograms by application for code refactoring, as shown in the following diagram.

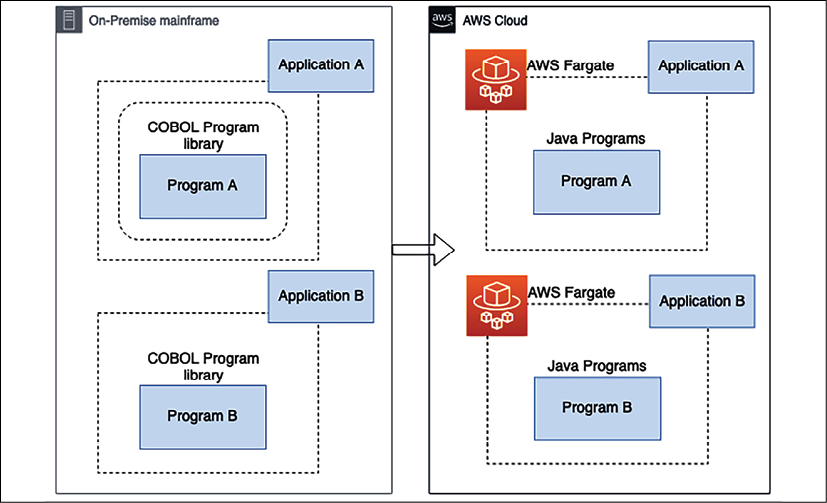


Figure 17.7: Mainframe modernization for a standalone application

In the preceding diagram, the mainframe programs and subprograms are written in COBOL, and the code is migrated to Java on AWS. However, you can use these decoupling patterns with your programming languages of choice. The migration pattern is legacy automated refactoring, where code, data, and dependencies are automatically converted to a modern language, data store, and framework while guaranteeing functional equivalence with the same business functions. Refactoring involves using automated tools to convert the mainframe programming language (such as COBOL) into modern programming languages (such as Java or .NET).

Refactored applications are deployed on containers that are provisioned and managed by **AWS Fargate**. Fargate is a serverless compute engine for containers that work with both Amazon **Elastic Container Service** (**ECS**) and Amazon **Elastic Kubernetes Service** (**EKS**). Here, mainframe database tables and mainframe files are migrated with the application.

After grouping, you can migrate applications A and B in the same wave or in different waves. In either case, for each application, package the refactored modern components and deploy them together into a runtime environment. After migration, retire the on-premises mainframe applications and their components. Now let's look at more complicated scenarios where code is shared by multiple applications.

**Migrating applications with shared code**

Let's assume mainframe applications A and B run shared code called program AB. You need to perform an impact analysis of the shared program AB to migrate applications A and B and program AB together. Based on the impact analysis, identify the number of dependent applications that use shared programs, such as program AB. You need to complete a business domain analysis to determine whether the shared program can be aggregated into a domain with applications and exposed as an API as one of the domain services. Let's look at some approaches you can take to decouple the applications in preparation for migration.

**Application decoupling using a standalone API**

Using this approach, you instantiate a standalone API by converting the shared COBOL program AB into a Java program. You can use automated refactoring tools provided to generate network APIs for the program to minimize refactoring efforts. You can take this approach when the shared program can be instantiated as a standalone service. The remaining components of applications A and B are refactored into Java as a whole and migrated to the cloud. You can migrate the applications in the same wave, as shown in the following diagram:

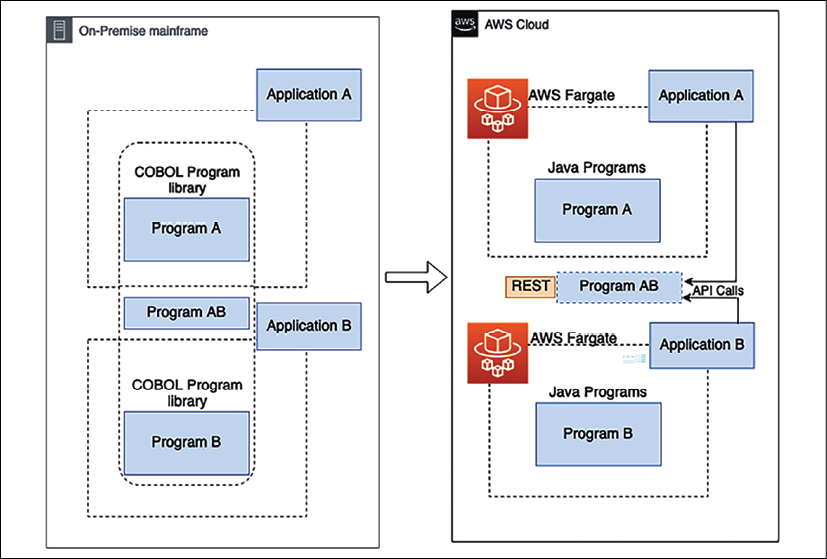


Figure 17.8: Migration of shared program applications using a standalone API

In this approach, you need to refactor both applications with their respective programs and migrate them to the cloud. You need to use the impact analysis report from the analysis phase to help developers and teams identify the refactored applications that call the shared program AB. Replace the inner program call to the shared program AB with network API calls. After the migration, retire the on-premises mainframe applications and their components.

**Application decoupling using a shared library**

In this approach, the shared program AB is converted into a Java common library and packaged with the migration applications. You should take this approach when the shared program is a supporting library instead of a standalone service. The remaining components of applications A and B are refactored into Java programs and migrated to the cloud.

This approach refactors applications A and B with their associated programs into Java and migrates them to the cloud. You should maintain the source code of the applications in a fully managed source control service such as AWS CodeCommit. The teams that use the shared program can collaborate on code changes by using pull requests, branching, and merging and can control the changes made to the shared program code. After the migration, retire the on-premises mainframe applications and their components.

When applications are too big to be grouped into the same migration wave, you can migrate them in multiple waves and maintain service continuity during the migration. With this approach, you can modernize your applications in phases without bundling them together. Migrating your applications in separate waves decouples them without requiring significant code changes in the mainframe.

**Application decoupling using message queues**

In this approach, the shared program AB is converted into a Java program and migrated to the cloud as part of application A. A message queue is used as an interface between the refactored application in the cloud and the legacy application on-premises. Using this approach, you can break up tightly coupled mainframe applications into producers and consumers and make them more modular to function independently. The additional advantage is that you can migrate the applications in different waves.

You can take this approach when applications on the mainframe can communicate with the migrated applications in the cloud through a message queue. It would be best to make sure that the queuing architecture pattern meets the business requirements for the applications that reside on the mainframe because it involves re-architecting the existing applications.

You should take the message queue approach if applications that aren't part of the first wave require a longer time (six months or more) to be migrated to the cloud. When applications are too big to be grouped into the same migration wave, you can migrate them in multiple waves, as shown in the following diagram, and maintain service continuity during migration.

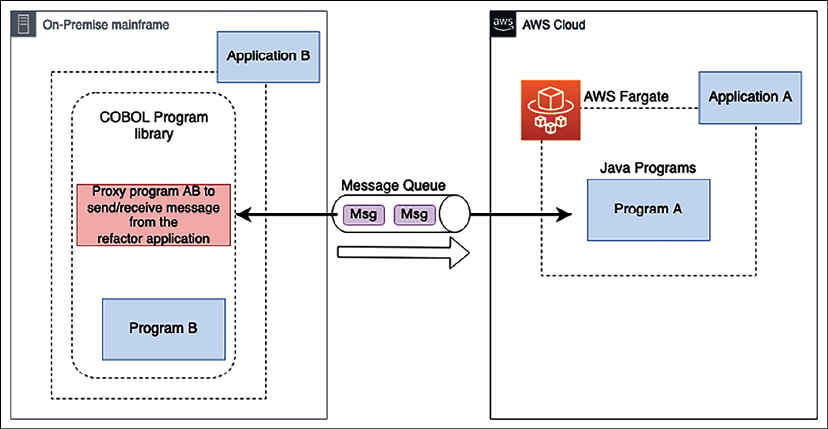


Figure 17.9: Migration of shared program applications using a message queue

As shown in the preceding diagram, you need to follow these steps for migration:

1. Migrate (refactor) application A with its associated programs to the cloud while application B continues to reside on-premises.
2. Refactor application A (in the cloud) to communicate with application B (on-premises) through a message queue.
3. Refactor application B on-premises to replace the shared program with a proxy program that sends messages to, and receives messages from, application A through the message queue.
4. After application A is migrated successfully, retire the on-premises application A and its components (including the shared program). Application B and its components continue to reside on-premises.
5. In the next set of migration waves, migrate application B and its components. The loosely coupled queuing architecture continues to act as an interface between applications A and B in the cloud. This reduces the refactoring effort for application B without impacting application A.

As a best practice, you should perform code analysis to produce a dependency map for the mainframe applications and identify the list of programs that are shared by applications. After that, group applications that share the same programs for the same migration wave to reduce program calls between the on-premises environment and the cloud. In the planning stage, run an impact analysis to identify applications that share programs with the application you're planning to migrate and select the right decoupling patterns for application migration. When possible, perform mainframe migration incrementally to reduce complexity and risk. By doing incremental migration, migration teams can provide faster feedback regarding the migration progress, and businesses can use the feedback to optimize internal processes to accelerate the pace of migration.

**Summary**

In this chapter, you learned about various challenges with legacy applications and why it is essential to modernize them. You learned about the different benefits an organization can get by upgrading its application to the latest technology. Application modernization can be a complicated and risky task, but is often worth the effort.

The outcome you get from the upgrade is a trade-off against the amount of investment and energy you put into it. Before defining the modernization approach, it's essential to understand your legacy system thoroughly. You learned various assessment attributes of an application in the aspects of technology, architecture, and code.

After the assessment, the next step is to define the modernization approach. You learned about various modernization approaches, including architecture-driven, system re-engineering, and migration approaches. You also learned about multiple techniques of system modernization, including straightforward approaches (encapsulation and rehosting) and complex approaches (rearchitecting and redesigning). The cloud can provide a significant value proposition, and you learned about the decision-making approach you need to take with modernization in the cloud.

You focused on the various technical aspects of solution architecture; however, documentation is one of the critical elements of architecture design to keep your system maintainable in the long run. The next chapter will discuss the documentation required for a solution architect to prepare, contribute to, and maintain maximum business value.

**Join our book's Discord space**