RED HAT®
CONSULTING

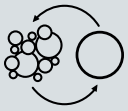
redhat.

WHITEPAPER

MAKING OLD APPLICATIONS NEW AGAIN

Software development patterns and processes for continuous application modernization

Zohaib Khan



"Let's face it, all we are doing is writing tomorrow's legacy software today."

MARTIN FOWLER
THOUGHTWORKS CHIEF ARCHITECT

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com

EXECUTIVE SUMMARY

One thing is clear when you look back at the past decades of IT and software engineering: everything changes. Periods of gradual improvement in hardware, language, infrastructure, and methodology are punctuated by paradigm-shifting innovation.

This evolution has allowed IT to stay ahead of ever-changing business demands, but it has not been easy or cheap. Many IT budgets are consumed by maintaining the old stuff and staying current with upgrades and migrations can deplete funding and resources before business benefits are realized.

With the right approach, it is possible to modernize a portfolio of applications in a way that yields value quicker and at lower cost—making it easier and less expensive to stay current as products and technologies continue to evolve.

In this whitepaper, Red Hat takes a look at three specific software development patterns to modernize existing applications. These modernization patterns address transitioning existing applications to more modern architectures and infrastructure and making them accessible to new applications. This paper also examines the conditions that lead to rewriting when that is the only option. These patterns help enterprises figure out how to get the most out of existing applications and establish a practice for continuous modernization that will serve the business now and in the future.

APPLICATION DEVELOPMENT AND DEPLOYMENT

Not so long ago, applications were coded in programming languages and compiled into a format that was unique to a processor and operating system. The applications were generally self-contained, tended to be large, and ran in private datacenters. Everyone assumed they would have long lifespans. These were built using heavyweight software development life-cycle approaches with formal, upfront requirements and long development timelines.

All of that has changed. Those applications are now called monolithic legacy applications—the dinosaurs of business applications. Although they served the purpose for which they were built, the pace of business and technical innovation accelerated, and these applications became a burden on the enterprise.

That innovation has led to the application development and deployment model commonly used today; DevOps processes that guide the creation of microservices, deployed on containers running in clouds. Just consider the progress in each of these four areas of application development: methodology, architecture, deployment, and infrastructure.

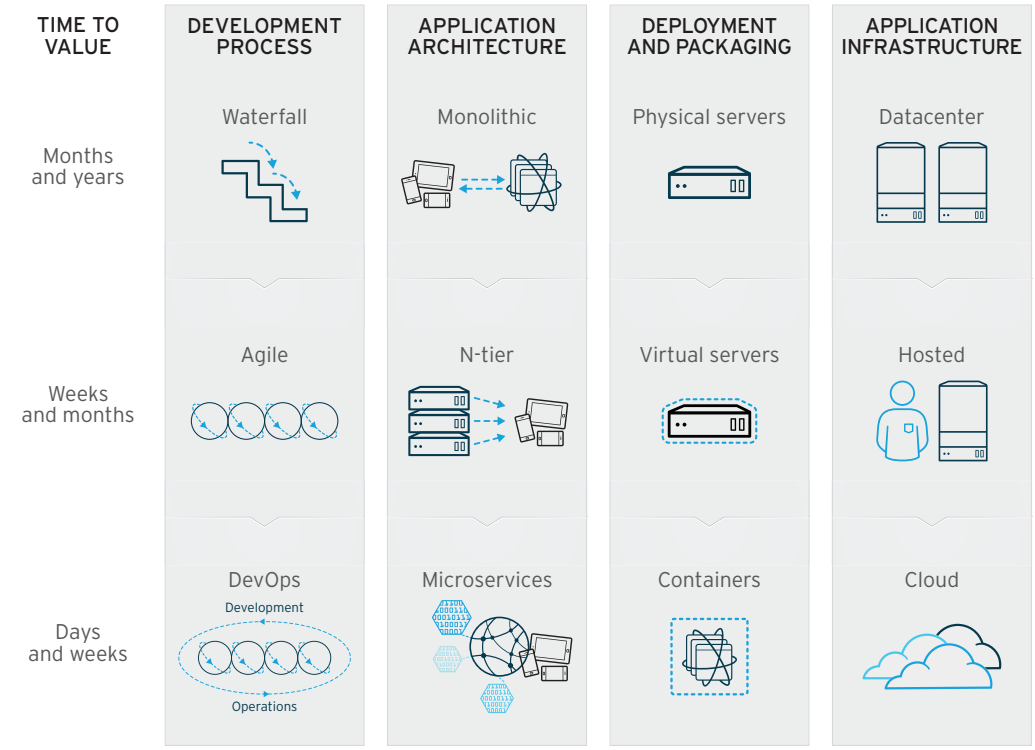


Figure 1: Evolution of application development and deployment.

A full solution stack includes all the components needed to develop and deploy an application, the methodology used to develop it, and the hardware it runs on.

The development process has gotten much faster. It evolved from a waterfall methodology with upfront requirements and a long time from specification to release, to iterative methodologies with frequent releases of incremental functionality, and now to highly collaborative DevOps practices with automated and continuous integration and delivery.

The trend in application architecture is the separation of functionality into components. All-in-one applications were decoupled into separate tiers for interface, logic, and data. Service-oriented architecture allowed applications to be built using dedicated services over a common enterprise service bus (ESB). This continued through to the current model based on microservices and application programming interfaces (APIs), where highly specialized components can be accessed by other services and applications. Microservices and APIs build on previous standards of web services to separate interface (how they are accessed) from implementation (how they do what they do) and can be implemented in a variety of languages and deployed on different systems.

Application deployment has become much more flexible. Applications are no longer tightly coupled to hardware. They are written to standards, such as Java™ Platform, Enterprise Edition (Java EE) so they can be deployed on many combinations of hardware and operating systems. Virtual machines and containers also allow applications to be packaged and deployed more easily to different hosts.

Application infrastructure evolved from large application-specific servers to horizontally scaled commodity servers supporting many applications. It is common now for applications to be deployed on multiple servers in dispersed datacenters, private clouds, and public clouds. This is much faster to deploy, and it improves performance and availability.

The evolution in application development and deployment across these four areas has led to faster initial development, more frequent updates, higher quality, closer alignment to business needs, greater flexibility in operations, and reduced costs.

Modernization is not about adopting new technologies and practices, it is about what happens to the old ones.

MODERNIZATION

These days, we refer to the software components supporting an application as a stack. More broadly, a full stack includes all the components needed to develop and deploy an application, the methodology used to develop it, and the hardware it runs on.

Companies have embraced new products and technologies for new projects, but legacy solution stacks are still used at many companies.

For example, many financial services firms developed custom applications decades ago. They deployed Bloomberg terminals and trader workstations, client-server systems, and n-tier web applications. Today, those same firms are creating mobile applications for customers and employees. As a result of growth and acquisition, they are running dozens of modern and legacy stacks throughout their application portfolio.

Modernization is not about adopting new technologies and practices, it is about what happens to the old ones. Imagine an old house heated with coal with newer sections heated by oil and gas. Upgrading the entire house to solar is expensive with little return, but it makes sense to use solar in the latest addition, and it is worthwhile to make it all work together under one roof.

With the right approach to modernization—the right methodology and patterns—enterprises can embrace and excel with change.

There are two primary goals of application modernization: use existing functionality and data in new applications as much as possible (deriving new value from old applications), and bring the benefits of new processes, products, and technologies to old applications.

THREE SOFTWARE DEVELOPMENT PATTERNS FOR MODERNIZATION

The following three software development patterns provide a unified approach to modernizing applications. Two extend the life and utility of existing applications and avoid all-at-once changes to the full portfolio, setting the stage for the third pattern, which covers gradual application refactoring and architectural updates. In no case is it necessary to rewrite an application all at once to benefit from modernization. Even monolithic applications can benefit from modernization without being rewritten.

LIFT AND SHIFT

Lifting and shifting modernizes how existing applications are packaged and deployed. By lifting and shifting, existing components are deployed on a modern deployment platform. A familiar example is application virtualization, where the application is packaged with the operating system and run as a virtual machine instead of on dedicated hardware.

Lifting and shifting is not intended to modernize the application architecture. Instead, it gets enterprises running on a modern deployment platform with a buffer of time to refactor the application later.

Lifting and shifting can be used to improve application performance by deploying on current and faster hardware. Applications become more flexible with simple deployment processes on modern platforms. Operational costs may be reduced too by retiring one-off servers and centralizing management.

Here are some common examples of lifting and shifting:

> A three-tier application with presentation, business logic, and data services has each tier running on a different Linux® server. Each service is repackaged as a container that includes all the configuration and runtime dependencies of that service. The containers are deployed in a Platform-as-a-Service (PaaS) environment running on a public cloud.
>
> A content management system (CMS) built on J2EE and running on four virtual machines is repackaged as a set of containers and deployed in a PaaS. In addition to the advantages of containers running in a PaaS, development teams benefit from an integrated developer experience that includes continuous integration and continuous delivery (CI/CD).

In these examples, the architecture is unchanged but the applications have new life on a modern deployment platform. This gives teams time to refactor the applications and modernize their architecture. For example, the four CMS components can be rearchitected as microservices and the entire CMS encapsulated as an API—making it accessible by mobile, cloud, and other applications that need content management capabilities.

Lifting and shifting is not appropriate for all applications. Applications that are locked into specific vendor solutions may be difficult to repackage and redeploy. Older operating systems may not be supported by the latest deployment platforms.

AUGMENT WITH NEW LAYERS

Many firms create business value by delivering applications over new conduits—such as mobile and Salesforce—and by integrating with partner applications. Augmenting with new layers is a software development pattern that can help make existing application functionality accessible to new applications and conduits, reducing development time and costs because the functionality does not have to be redeveloped. It is better to use complex and critical functionality where it exists, since it has been thoroughly proven over time.

Augmenting with new layers involves creating a new layer of application software that wraps the existing application functionality and data with an interface that is accessible to new applications. To avoid introducing excessive complexity, the layer usually has no extra business logic but simply serves as an adapter between the new and the old.

It is usually not necessary to modify the existing application, making this pattern appealing when the source code is not available or when it is too risky to modify the existing application.

As with lifting and shifting, the architecture of the existing application is unchanged but augmenting with new layers uses existing functionality to create new applications and services using a current architecture.

Augmenting with new layers sets the stage for gradual application architecture modernization. Over time, old application functionality can be rewritten and old stacks retired. This pattern can then be used to migrate slowly off an existing application (known as a strangler application), which is usually better than an all-at-once rewrite.

Here are some examples of augmenting with new layers:

> An existing commercial application runs on an operating system that is not supported by current virtualization or container platforms. It is accessible through a standard API. An adapter microservice is written that accesses this API. This microservice is then used by new applications and other microservices to access the commercial application's functionality.
>
> An ordering application written in Visual Basic makes extensive use of complex stored procedures in its database back end. A new mobile ordering application is being developed for phones and tablets. An adapter layer is created with a microservices interface to the new application. The adapter conceals the database and stored procedures. All new applications access the adapter, which converts the requests into calls to the stored procedures.

In both examples, augmenting with new layers is only an interim solution that lets new development with a current architecture use an existing application. This pattern is compatible with lifting and shifting, and they can be applied together. While one makes the functionality of existing applications available to new applications, the other makes existing applications easier and less expensive to deploy and manage, but both give developers time to refactor and rewrite existing applications using a current architecture.

REWRITE

Rewriting an application is different than creating new applications from scratch; it is the process of creating new functionality to replace and retire existing applications. As part of an overall modernization strategy, rewriting can follow lifting and shifting and augmenting with new layers, and it is the only way to update the application architecture for a fully modern stack.

Rewriting an existing application is usually the least appealing option of the three. It is likely expensive, time-consuming, and may take years for costs to offset. Unless the application delivers new business value, rewriting it is hard to justify to executives with limited budgets.

Nevertheless, there are cases where rewriting is the only option. There may be very old applications running on operating systems and hardware that are no longer supported by the vendors. Companies do not want to run their critical business systems without the safety net of vendor support, and sometimes no one has the skills to operate old systems.

When rewriting is the only way to go, the best approach is to migrate functionality off old applications gradually; augmenting with new layers can make that possible. It is also a good idea to delay rewriting because some functionality will become obsolete and will not need to be migrated at all. Resist the temptation to simply migrate old behaviors, and instead plan and prioritize as if developing a new application. This will help create applications that are more flexible and accommodating to the changes that will come.

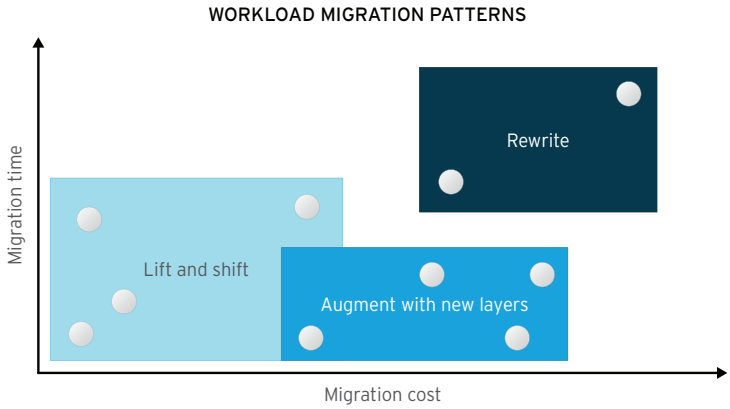


Figure 2: Workload migration patterns determined by time against cost. Benefits are not shown because they are dependent on the application and its current and future states.

PATTERN SELECTION

There is no single best pattern—it entirely depends on the application, the business, and contextual factors. The patterns address different areas of application development and deployment, so multiple patterns might apply to one application or hold no benefit at all. In general, modernizing a portfolio usually requires more than one software development pattern.

Lifting and shifting is usually the least expensive pattern to apply. Augmenting with new layers can be more costly because it requires a development effort, but it can often be applied more quickly. Rewriting is almost always the most expensive and time-consuming (Figure 2).

These software development patterns help enterprises figure out how to get the most out of existing applications and establish a practice for continuous modernization that will serve the business now and in the future.

A METHODOLOGY FOR MODERNIZATION

The three modernization patterns help application architects and development and operations managers bring new life to existing applications, but they are usually just part of a larger initiative. When this is the case, a formal methodology helps organize the effort, keep costs in check, and bring value to the business.

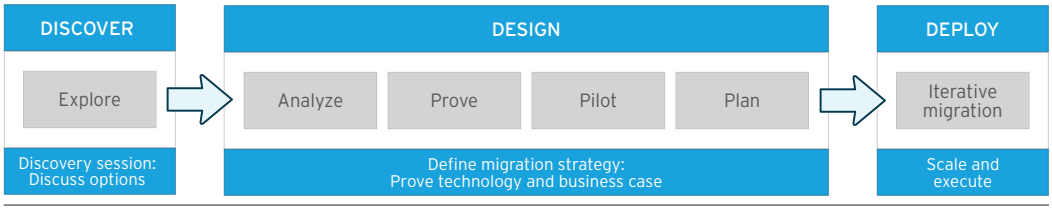


Figure 3: Red Hat's application modernization process.

Red Hat uses a multiphase, iterative process for application modernization (Figure 3). The main objective is to create a plan and execute it multiple times, where each iteration results in new, incremental value. This reduces the risk of ripping and replacing.

Interactive sessions during the discover phase identify current application states and objectives. Key stakeholders explore potential ways to modernize and begin prioritizing the steps with the goal of getting a commitment to proceed.

In the design phase, a series of steps take the process through analysis, proof of concept, and piloting. Automated application code analysis helps identify potential issues and estimate efforts while users dive into the stack and architecture to identify current states and determine modernization candidates. Desired states are designed in this step, modernization pattern are selected, and a detailed deployment plan is created.

The deployment plan is executed in the next step. Organized into multiple iterations to reduce risk and learn from previous steps, this model should be guided by a center of excellence staffed with migration specialists. This group captures best practices and ensures sustainability as the project evolves and stakeholders change.

CONCLUSION

Application modernization can seem daunting when applied to a large portfolio of applications, particularly when they run on older stacks and outdated hardware. It is hard to see past the time and effort it will take, and lowering operational costs is rarely a sufficient justification for funding.

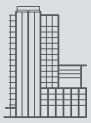
Red Hat's methodology breaks the problem down so that only a few applications are migrated at a time. Applications are evaluated based on the cost and time it would take to modernize, post-modernization operational costs, and potential business value.

The three modernization patterns provide several standard approaches to extend the life of existing applications and help determine when it is necessary to rewrite from scratch. With the right approach to modernization—the right methodology and patterns—enterprises can embrace and excel with change.

Red Hat® Consulting uses the patterns presented above to help organizations get more value from their applications. Learn more by visiting redhat.com/en/resources/application-migration-modernization-consulting-jboss-middleware-datasheet.

ABOUT THE AUTHOR

Zohaib Khan is currently the application migration practice lead and a manager of the PaaS community of practice at Red Hat. Zohaib has presented at various conferences, including Red Hat Summit, StackWorld, and Red Hat Technical Exchange. He has also written about DevOps and technology disruption and innovation. Prior to Red Hat, Zohaib was a senior manager in healthcare and financial services and experimented with entrepreneurship by co-founding a technology services company.



ABOUT RED HAT

Red Hat is the world's leading provider of open source software solutions, using a community-powered approach to provide reliable and high-performing cloud, Linux, middleware, storage, and virtualization technologies. Red Hat also offers award-winning support, training, and consulting services. As a connective hub in a global network of enterprises, partners, and open source communities, Red Hat helps create relevant, innovative technologies that liberate resources for growth and prepare customers for the future of IT.

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com
#0460201_1216

NORTH AMERICA
1 888 REDHAT1

EUROPE, MIDDLE EAST, AND AFRICA
00800 7334 2835
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
info-latam@redhat.com