Don’t just migrate, modernize

A guide to building modern apps to drive competitive advantage

In collaboration with

The modern world is a digital one, and businesses must meet customers wherever they live, work, buy, and sell. Organizations that still rely on legacy systems and processes lag behind those that leverage modern cloud technology and operational models to innovate applications faster and compete more effectively.

In this eBook, you'll discover how cloud migration is the first step in driving app modernization and identify the elements of modern apps. You'll learn how adopting DevOps creates a culture for competitive advantage and see how businesses of any size can modernize and still achieve speed, agility, and cost savings even with fewer resources or disjointed teams.

Have an AWS account?

Use self-paced workshops and training modules to build experimental apps as you follow this guide.

Why modernize?

Modernization is a business priority today. To delight customers and win new business, organizations need to build reliable, scalable, and secure applications that create big customer value. That means adopting new technologies and practices, such as serverless computing, microservices, continuous integration/continuous delivery (CI/CD), and containers. The transition can be tough, but the rewards are worth the effort.

50%

of information communication technology spending will be directly allocated for digital transformation

67%

of executives believe they must pick up the pace to remain competitive

90%

of new applications are predicted to be cloud-native by 2025

The benefits of modernizing

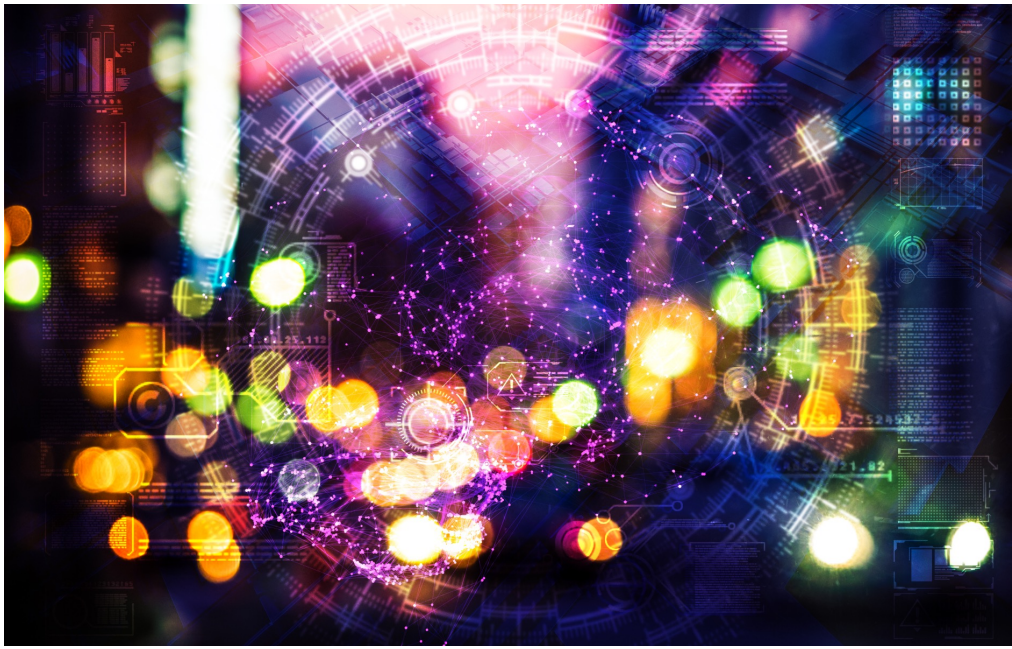
Modernizing is all about simplicity, agility, and delivering value. How? Automation. By automating as much of the development cycles as possible—including testing, deployments, provisioning, open-source security, and compliance processes—businesses are one giant step closer to modernization.

An organization's migration strategy should guide its teams to move quickly and independently. This is the promise of a DevOps operational model.

What is DevOps?

DevOps merges formerly siloed—and sometimes adversarial—development and operations teams to work together in both synchronous and asynchronous ways to deliver constant and impactful outcomes for customers. DevOps teams automate processes that historically have been manual and slow to deliver customer value. From its inception, DevOps has been a philosophy for working—not a conglomeration of tools or a software suite—and a driver for an organization's cultural transformation toward modernization.

Going all in on modernization and DevOps generates multiple benefits.



Faster time to market

By speeding up the release cycle and offloading operational overhead, developers can quickly build new features. Managing the entire software supply chain in a single secure system across your organization offers a single source of truth with wide observability. Automated test and release processes reduce error rates, so that products are market-ready faster.

Increased innovation

Developers are 1.5x more likely to feel innovative with a mature DevOps model. Automating the release pipeline through CI/CD helps teams release high-quality code faster and more often. It also reduces the time it takes to test and release so that teams have more time for innovation.

Improved reliability

By automating test procedures, scanning code in development for vulnerabilities and monitoring at every stage of the development lifecycle, modern applications are reliable at deployment because any issues can be evaluated and addressed in real time.

Reduced costs

With a pay-for-value pricing model, modern applications reduce the cost of over-provisioning or paying for idle resources. By offloading infrastructure management, maintenance costs are also lower.

Looking for more ways DevOps positively impacts an organization? Read the report on how DevOps gets development, security, and operations teams on one page.

The elements of modern applications

While IT faces waves of pressure to support the next leading-edge technology to meet customer demands, organizations must remain focused on getting or maintaining a competitive edge, and they must find modernization efforts that create value for the enterprise. That starts by understanding the five elements that compose modern applications.

1. Application architecture: Monolith versus microservices

Monolith

One of the most common ways to build enterprise applications is on a monolithic architecture as a single, unified application where all the components are tightly coupled and working from a shared database. Most frameworks today are built around monoliths such as Spring for Java, Ruby on Rails, Django for Python, or Express for node.js.

As monolithic applications grow, they require more resources, from compute and memory requirements to storage and network bandwidth. Although these issues can be solved by scaling the application servers vertically up or horizontally out, this approach naturally scales the *whole* application, even if only a single module requires the extra resources.

Monolithic applications are popular because they are fast to develop but difficult to scale and update as the code base grows because each aspect of the application is tightly coupled. These applications grow increasingly complex over time, complicating maintenance so much that even the smallest changes require significant effort for development, testing, and deployment, decreasing the business' agility.

Because of the challenges inherent in monolithic applications, many modern applications have shifted to a new paradigm, commonly known as a *microservices architecture*.

Microservices

Microservices are small services providing a bounded context of functionality—each potentially using their own data store—and predominantly integrating with other services through event-driven communication. Each service is designed for a set of capabilities and focuses on solving a specific problem. If developers contribute more code to a service over time and the service becomes complex, it can be broken into even smaller services.

A microservices architecture breaks a single-process application into multiple components that work together to deliver value. Any communication between individual components happens via well-defined, loosely coupled APIs, or through events and messaging. Each component service in a microservices architecture can be developed, deployed, operated, and scaled without affecting the functioning of other services. Services do not need to share any of their code or implementation with other services: they act as self-contained black boxes.

Breaking monolithic applications into loosely coupled microservices can help overcome many of the challenges presented by monolithic applications because each service can be independently deployed and scaled. By ensuring each microservice has its own development lifecycle, DevOps teams are no longer tied to other teams' release cycles, enabling them to accelerate their deployment frequency, improve their agility, and increase the business' ability to respond to change.

2. Software delivery: Automation, abstraction, and standardization

Front- and back-end developers need tools, frameworks, and processes that enable them to rapidly and securely deliver new features to customers—often required on a daily or hourly cadence. Automated release pipelines, including CI/CD, enable teams to rapidly scan, test, and release lots of code while minimizing errors. Frameworks and tooling abstraction eliminate the complexity of provisioning and configuring resources. Standardization through infrastructure-as-code templates provision the entire technology stack for an application through code, ensuring DevOps teams meet central requirements.

3. Data strategy: Decoupled and purpose-built

Much like a monolithic application, a single database is also difficult to scale. It can become a single point of failure with fault tolerance challenges. Modern applications take advantage of decoupled data stores where there is a one-to-one mapping of database and microservice. By decoupling data and microservices, DevOps teams are free to choose the database that best fits the needs of the service, like choosing a database that is purpose-built for the task at hand.

4. Operations: As serverless as possible

Modern applications have a lot of moving parts, including many microservices with unique databases that release features often. Serverless technologies reduce that support burden because they run without the need for infrastructure provisioning and scaling and have built-in availability and security. Plus, they have a pay-for-value billing model. There are serverless services for the entire application stack: compute, storage, and integration.

5. Management and governance: Programmatic guardrails

Managing an organization securely, legally, and safely is priority one, but strong governance often results in checkpoints that delay innovation. Increasingly, organizations address this by adopting the concept of guardrails—mechanisms such as processes of practices—that reduce both the occurrence and blast radius of undesirable application behavior. Usually expressed as code, guardrails can standardize processes and practices for the monitoring, provisioning, deployment, cost management, and security of applications, without creating bottlenecks or slowing innovation.

Remember:

Modernization is about simplifying too. As their architectural patterns and software delivery processes change, an organization should adopt an operational model that enables them to offload any activity that isn't a core competency.

To gain agility that can enable rapid innovation, AWS recommends building a microservices architecture, operating and deploying software using automation for things like monitoring, provisioning, cost management, deployment, and security and governance of applications.

Choosing a serverless-first strategy—opting for serverless technologies wherever possible—enables businesses to maximize the operational benefits of AWS and AWS Partners. A serverless operational model allows them to:

- Build and run applications and services without provisioning and managing servers.
- Scale flexibly.
- Pay only for value.
- Automate high availability.

A serverless model also lets organizations build and manage aspects of their application that deliver customer value without having to worry about the underlying detail.

Whether developing net-new applications or migrating legacy ones, building with serverless primitives for compute, data, and integration enables businesses to benefit from the most agility the cloud has to offer.

Adopting microservices architectural patterns doesn't have to be an all-or-nothing endeavor. There are two common paths to a service-oriented architecture:

A. Wrap the existing monolith in APIs and treat it as a black box while building net-new functionality as microservices.

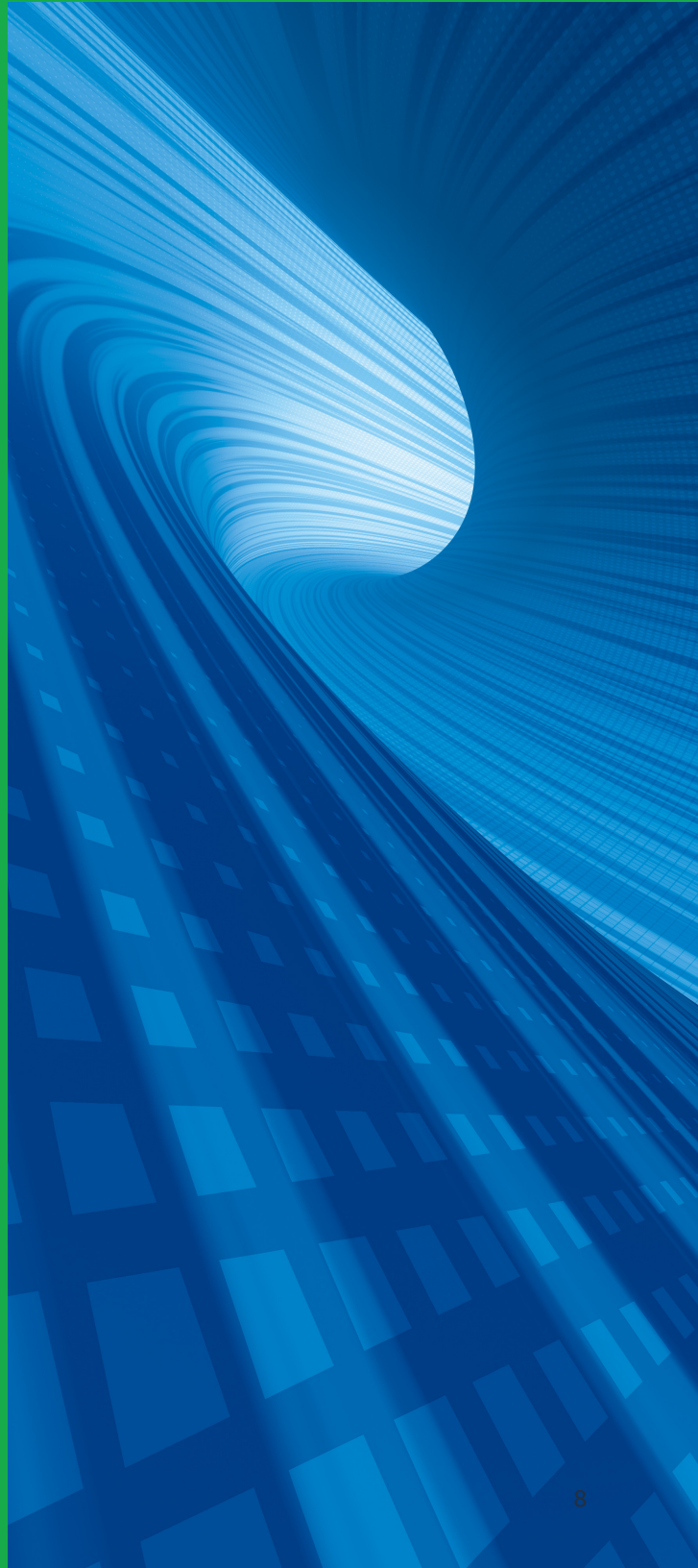
B. Refactor the monolith to microservices using the strangler pattern, where development teams carve out functionality that is already fairly decoupled from the monolith.

Both avenues have benefits and downsides, but both require first setting up the appropriate development infrastructure. This includes building automated software delivery pipelines to independently build, test, and deploy executable services, plus, the infrastructure to secure, monitor, operate, and debug a distributed system.

Keeping the monolith as-is can work if it's a standalone system that won't require updates to its core functionality. In this case, most new development effort can go to building new microservices that simply connect to the monolith through APIs.

If the monolith can't be maintained or thrown away, but some of its parts need to be rewritten, using the strangler pattern is the best approach. This functionality is decoupled from the monolith behind an API for easy replacement and decommissioning once the microservice is built. This means capabilities that don't require changes to many client-facing apps—and potentially don't need their own data store—are ideal first candidates.

For example, in an e-commerce application, a few potential services to consider are authentication, invoicing, or customer profiles. When carving out their first few microservices, most DevOps teams aim to test and optimize their software delivery pipelines and API approaches and upskill team members, rather than optimizing for functionality.



Don’t just migrate, modernize. A step-by-step guide

The proliferation of fast, affordable computing has allowed organizations of all sizes to create internal efficiencies and reach more customers through digital products. However, the ubiquity of tools, multiple paths to market, and changing consumer preferences mean businesses in every industry must innovate faster than ever to remain competitive.

Modern applications that are built on a microservices architecture enable and accelerate innovation by distributing the effort and investment over time and across smaller teams—automating and increasing the speed of testing and delivering changes to the market. Modern applications allow fine-grained resource optimization and enable DevOps teams to rapidly scale both how they *build* products and how they *run* them.

Here’s how to start modernizing beyond migration.

STEP 1

Migrate the monolith

When responding to a compelling event—for example, Data Center Exit—start by quickly migrating an existing monolith (Java Spring Boot) to the AWS Cloud using a variety of tools and patterns, including lifting and shifting with AWS Elastic Beanstalk and containers.

Outcome
A functioning, scalable cloud-based application

STEP 2

Build application release automation and package management

Once the business has responded to the compelling event, the team should focus on building application release automation with package management, and they should adopt DevOps and agile practices. To create speed and agility, they should start to identify common CI/CD patterns and implement these on AWS or with AWS Partner products such as JFrog Artifactory, which serves as the single source of truth for all packages, container images, and Helm charts as they move across the entire DevOps pipeline.

Outcome
Release pipelines that are fully automated, robust, and repeatable, enabling faster releases and increased agility to serve customers

STEP 3

Create microservices

Identify bounded contexts and apply common migration patterns (such as the strangler pattern) within a monolithic architecture to create microservices.

Outcome
An established foundation to start decoupling and creating microservices, while the application is still fully functional

STEP 4

Refactor data to microservices

Modern applications take advantage of decoupled data stores where there is a one-to-one mapping of database and microservice.

Outcome
Decoupled data and microservices that enable DevOps teams to choose the database that best fits the needs of the service

STEP 5

Implement microservice messaging and event-driven architectures

Allow microservices to communicate by implementing messaging and event-driven architectures. Focus on common eventing patterns such as Event Notification, Event Carried State Transfer, Event Sourcing, and CQRS. Apply these patterns during the modernization journey.

Outcome
Technology independence, established event-driven architectures, and communication between decoupled microservices

STEP 6

Implement API-based microservice authentication and authorization

Enable common authentication and authorization patterns with microservices including technology like OAuth, Bear Tokens, and JWT.

Outcome
Secured services with role-based access control (RBAC)

STEP 7

Initiate hackathons, workshops, and bug bounties

Share the newly gained knowledge with the rest of the organization. Create internal programs like hackathons, workshops, and bug bounties to build collaboration and continue driving the modernization strategy.

Outcome
Internal champions and centers of excellence that help accelerate the modernization journey

CASE STUDY

Monster

Situation

Every day for 25 years, Monster—a global leader in connecting people and jobs—has worked to transform the recruiting industry. Today, the company leverages advanced technology including intelligent digital, social, and mobile solutions, its flagship website Monster.com, Monster's innovative app, and a vast array of products and services to match employers and candidates.

Challenges

After 20 years of market leadership, Monster was ready to refresh core technologies that hadn't kept pace with market changes. Legacy applications built on monolithic architectures required many months to release even a single new feature into production, which slowed innovation. **"The dependencies between some of the components was so extreme,"** recalled Martin Eggenberger, Monster's chief architect. **"We just didn't have any clear idea what was running there."**

Storing artifacts in cloud buckets made versioning, labeling, and promotion very challenging. **"It became very, very obvious rather quickly that we had to go for a complete digital transformation of the Monster engineering organization,"** Eggenberger remarked.

This would require a major reinvention of its core technologies to be cloud-native, built around containerized microservices architectures and Kubernetes.

Solution

Monster uses AWS for its operational systems and self-manages a high-availability installation of the JFrog Platform in an EKS cluster in its home region. Ninety developers from 15 globally dispersed teams build core applications on the Java Spring Framework, with additional development for NodeJS, Python, and other language environments.

Results

JFrog Artifactory enables an omniverse of polyglot development with secure local repositories for Maven, npm, PiP, Docker, Helm, and more. Artifactory's virtual repositories unify local package management with cached access to remote resources like Maven Central and Docker Hub, while JFrog Xray helps Monster stay vigilant against vulnerable components and builds before they reach production.

Monster promotes cloud-native builds using JFrog Artifactory repositories for development, testing (including JFrog Xray vulnerability scanning), and staging. Spinnaker pulls from private Docker registries in JFrog Artifactory to deliver services into Kubernetes clusters in AWS and elsewhere for canary testing and production.

Instead of a 15-month cycle, Eggenberger boasts that **"today, Monster can release virtually on request."**

Learn more now

- **Explore** the AWS Builders' Library
- **Attend** an AWS modernization workshops
- **Visit** JFrog on the AWS Marketplace

aws marketplace