AWS Prescriptive Guidance
Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway
AWS Prescriptive Guidance: Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway

Copyright © Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

Introduction ...................................................................................................................................... 1  
  Targeted business outcomes ........................................................................................................ 1  
Types of ASP.NET web services ........................................................................................................ 2  
The strangler fig pattern ..................................................................................................................... 3  
REST-based ASP.NET web services .................................................................................................... 4  
SOAP-based ASP.NET web services (ASMX) ..................................................................................... 6  
  Example architecture .................................................................................................................. 8  
Other considerations .......................................................................................................................... 9  
  Authentication and authorization ................................................................................................. 9  
  Domain-driven design .................................................................................................................... 9  
  Interim states and target state ...................................................................................................... 9  
FAQ ................................................................................................................................................ 10  
  Does AWS support Windows containers? ..................................................................................... 10  
  Can I run ASP.NET services on Linux? .......................................................................................... 10  
  Can I run Windows containers on AWS Fargate? ....................................................................... 10  
  Does Amazon API Gateway support Windows-integrated authentication? Does it support OAuth 2.0? ... 10  
Next steps and resources ................................................................................................................... 11  
AWS Prescriptive Guidance glossary .................................................................................................... 12  
Document history ............................................................................................................................. 20
Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway

Dan Broadway, Senior Modernization Architect, AWS Professional Services

November 2021

This guide describes approaches and best practices for implementing the strangler fig pattern to modernize legacy ASP.NET web services on Amazon Web Services (AWS). The strangler fig pattern is a modernization approach in which the existing functionalities of a legacy system are incrementally replaced with new applications and services until the new system eventually replaces all the old or key parts of the legacy system. As functionality is replaced, the inactive parts of the legacy (and often monolithic) system can be removed or retired.

The guide discusses a modified branch by abstraction approach, which is a common technique for incrementally changing software systems, combined with service delegation, where a service defers its behavior to another, newer service, to help manage breaking changes to service consumers.

Targeted business outcomes

The intent of this guide is to help customers modernize legacy ASP.NET services that are based on the Representational State Transfer (REST) or Simple Object Access Protocol (SOAP) standard on AWS. It focuses on scenarios where an incremental modernization approach is required in order to manage the delivery risk that is often associated with decoupling larger, monolithic applications. It also discusses scenarios where services with multiple consumers cannot be upgraded in synchrony without introducing breaking changes.

This guide is intended to help you:

- Migrate, modernize, and scale your ASP.NET applications on AWS
- Manage the delivery risk associated with modernizing legacy applications
- Decouple services that have multiple, highly available consumers
Types of ASP.NET web services

When you’re planning to modernize your legacy ASP.NET web services on AWS, you can generally group these services in two categories:

- REST-based ASP.NET web services. These are built using either ASP.NET Web API (RESTful) or the ASP.NET Model-View-Controller (MVC) pattern.
- SOAP-based ASP.NET web services. These are implemented by using Windows Communication Foundation (WCF) or ASP.NET Web Service (ASMX) files and ASP.NET WebMethods.

REST is the dominant architectural style in use today for web APIs. Therefore, services that are implemented using this approach are compatible with most modern tooling and frameworks, so they have a more straightforward modernization path. Modernizing these services is facilitated using API gateways and API managers, and primary concerns involve how the services model the resources and business domains they represent, and how the services are secured. However, web services that rely on SOAP generally predate REST-based services, and, therefore, have limited support today. Migration of these services involve the same concerns as REST-based services. They also face the additional challenge of little to no tooling and framework support.
The strangler fig pattern

The strangler fig pattern was introduced by Martin Fowler as a way to manage risk when modernizing or rewriting large, monolithic systems. The pattern is an analogy for a type of plant that begins life as a vine growing alongside an older, established tree. As the vine grows, it spreads to completely consume and ultimately replace the host tree, leaving a new, strangler fig tree in its place. In the context of modernizing ASP.NET web services, this pattern incrementally replaces system functionality by establishing proxies where other systems have dependencies on web services. Initially, you can consider these proxies as having pass-through behavior, because their implementation is fulfilled by the existing monolithic application service. In the natural analog, this is when the strangler fig initially sends a vine up the host tree’s trunk. Then, a new service, which is decoupled from the monolith, is created, and the proxy’s implementation is deferred to that new service. In the natural analog, this is when the strangler fig vine wraps around one of the tree’s branches and overtakes it. This pattern of proxying and then swapping out the proxy implementation with a new service continues until all the legacy system’s functions are migrated to new services. At this point, the strangler fig vine completely consumes the tree, and the legacy system can be decommissioned.

Follow these best practices when using the strangler fig pattern, so you can independently scale and deploy your application more smoothly:

- Select a component that has good test coverage and less technical debt associated with it. Starting with this component can give teams a lot of confidence during the modernization process.
- Select components that have scalability requirements, and start with one of these components.
- Select a component that has frequent business requirement changes and frequent deployments.
- To implement this pattern at scale on AWS, deploy the refactored ASMX services in a Windows container that is running in Amazon Elastic Container Service (Amazon ECS), and publish your modernized REST API by using Amazon API Gateway.
REST-based ASP.NET web services

When you modernize REST-based ASP.NET services on AWS by using the strangler fig pattern, we recommend that you use Amazon API Gateway to establish the proxy that will be used to divert traffic to the new service. You can introduce an API Gateway endpoint as the intermediary between service consumers and the legacy service that is being modernized. If the legacy service is already on AWS, the API Gateway endpoint is configured to route requests to the legacy REST service. If the service is not yet on AWS, it can be migrated as is before establishing the new API Gateway proxy. If that isn’t possible, you can take a hybrid cloud approach by using an AWS connectivity service such as AWS Direct Connect to connect API Gateway to your on-premises data center. The following illustration depicts the ASP.NET REST service and its consumer before and after the introduction of API Gateway as a proxy between the two.

Before the introduction of a proxy:

![Before the introduction of a proxy](image)

After adding API Gateway as a proxy:

![After adding API Gateway as a proxy](image)

When the API Gateway proxy is in place, you can create and deploy the modernized service on AWS by using Amazon ECS, for example, to achieve a highly scalable and available service. When the proxy and newly modernized service have been created and tested, you can reconfigure the API Gateway endpoint to point to the modernized REST API for its implementation.

![When the API Gateway proxy is in place](image)

If the newly modernized service has an API contract that is different from the legacy proxy contract the consuming systems depend on, you can use API Gateway’s data transformation feature. Incoming API
requests that are structured using the legacy system's schema can be mapped and transformed to the new service's contract.
SOAP-based ASP.NET web services (ASMX)

When you modernize a legacy ASP.NET web service (ASMX) to a REST API on AWS, it might have dependent consumers that can't, and most likely shouldn't due to the associated risks, be upgraded at the same time. The following illustration depicts an ASMX web service that has three dependent consumers.

Because these consumers call the ASMX service by using the SOAP protocol, replacing the ASMX service with a REST API would cause these consumers to break. Furthermore, refactoring the ASMX service to a REST API while you convert the system from a monolithic architecture to microservices will most likely involve remodeling the system behavior and domain model. This is likely to create additional breaking changes for consumers.

To maintain backward compatibility with the old service so that the system can be incrementally modernized, you can implement the guidance of the strangler fig approach. In the strangler fig approach, risks associated with modernizing an important legacy system are, in part, managed by gradually replacing the legacy system with a new one.

Branch by abstraction is another approach introduced by Martin Fowler that is an effective technique for implementing the strangler fig pattern. As with the strangler fig approach, branch by abstraction helps engineers make incremental and gradual changes to systems when major pieces of functionality have to be replaced or modernized. In this approach, an abstraction layer is introduced between the code that implements specific functionality and the code that consumes that functionality (that is, client code). This abstraction layer doesn't have to be an abstract type that can be inherited by concrete classes. Instead, it can be any implementation that captures the functional contract between the consumer and the provider.

In the case of legacy SOAP-based ASP.NET web services, you can enable the gradual migration of service consumers from old to new by maintaining compatibility, through an approach that is inspired by the branch by abstraction technique. However, the approach for SOAP-based ASP.NET web services uses
delegation instead of inheritance. Compatibility with ASMX consumers is achieved by refactoring the legacy ASMX service to delegate its implementation to the new REST API. The following illustration depicts the legacy ASMX service that is delegating its implementation to the new REST API. This delegation allows service consumer 3 to upgrade to the new REST API while service consumers 1 and 2 are upgraded independently.

The nature of the ASMX service refactoring depends on the functional and non-functional (cross-functional) requirements of that service. In some cases, the refactoring might involve translating from the old to the new authentication and authorization mechanisms, transforming the service ASMX payload from XML to JSON, and then invoking the REST API with the transformed payload. In more complex scenarios, the refactored ASMX service might have to orchestrate calls to multiple REST APIs and maintain state.

The process of migrating service consumers from the legacy ASMX service to the modernized REST API continues incrementally until all service consumers have been updated to reference the new API. The following illustration depicts the fully modernized state, where all service consumers reference the REST API, and the ASMX service has been decommissioned because it is no longer needed.
Example architecture

The end-state architecture for the SOAP-based ASMX service shows all service consumers using the modernized REST API, leading to the decommissioning of the legacy ASMX web service. However, the focus of this pattern is not the end-state architecture, but the interim-state architecture needed during migration, while consumers are being upgraded. This interim state is illustrated in the following diagram.

In this interim architecture, service consumers 1 and 2 are still using the legacy ASMX service, which has been refactored to delegate its implementation to the modernized REST API. The refactored Windows service is deployed as a Windows container in Amazon ECS, configured across three Availability Zones for high availability, and accessed through an Application Load Balancer. Service consumer 3 has been updated to use the new REST API directly through API Gateway.
Other considerations

So far, we’ve discussed using API Gateway and Windows containers to modernize your legacy ASP.NET web services on AWS. Here are some other considerations to take into account:

- Security
- Remodeling service domains
- Sequencing web service upgrades when there are many services to modernize

These are discussed in the following sections.

Authentication and authorization

Modern APIs generally leverage token-based (JSON Web Token or JWT) authentication and authorization standards such as OAuth 2.0 and OpenID Connect, whereas legacy ASP.NET web services traditionally relied on either basic authentication or Windows-integrated authentication to a Windows Active Directory domain. As a best practice, in cases where the old and new authentication and authorization approaches are incompatible, we recommend that you upgrade these security mechanisms in place before you modernize your web service on AWS. Attempting to map identities or trying to securely transfer security state between the old and new systems might not be a significant effort, but it might introduce security vulnerabilities.

Domain-driven design

When breaking a monolith into individual services, domain-driven design (DDD) is a best practice that is often used to model systems into cohesive business domains. DDD is an approach to developing software for complex needs by connecting the implementation to an evolving model of the core business concepts. Its premise is to place the project’s primary focus on the core domain and domain logic, and to base system designs on a model that reflects the business. If you use DDD when you are modernizing an existing, monolithic application, you need to work backward from the features and user flows of the monolith. You can use DDD in combination with the strangler pattern to guide the process of breaking the monolith and determining which services to strangle, hence the term domain-driven design.

Interim states and target state

When you're modernizing more than a few ASP.NET web services on AWS, it is a good practice to first define what your target state architecture will be after all services are modernized. However, the target state architecture isn't necessarily the end state or the final state, because business contexts change often and the system target state should be updated as needed to stay aligned with the business goals. When you’ve defined the target state, you can work backward from there to define interim-state architectures that incrementally fulfill the target state vision. You can think of these interim-state architectures as the progression of the strangler fig vine up the tree as it encounters and engulfs major portions of the host tree. Interim-state architectures often introduce temporary or overlapping constructs that will not be part of the end-state architecture in order to facilitate the evolution to the target state. The modernization of the SOAP-based ASP.NET web service provides an example of this: A Windows-based container is introduced temporarily to bridge the gap between calling systems that are dependent on the legacy service until they can be upgraded to the new REST API.
FAQ

This section provides answers to commonly raised questions about modernizing legacy ASP.NET web services.

Does AWS support Windows containers?

Yes, both Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS) support Windows containers. To get started with these AWS services using Windows containers, see the following:

- Getting started with the Amazon ECS console using Amazon EC2 Windows containers (Amazon ECS documentation)
- Amazon EKS Windows Container Support now Generally Available (AWS News blog)

Can I run ASP.NET services on Linux?

Yes, ASP.NET services (APIs) written in .NET Core can run on Linux. To run your .NET Core applications on Linux by using AWS Elastic Beanstalk, see Working with .NET Core on Linux in the Elastic Beanstalk documentation.

Can I run Windows containers on AWS Fargate?

As of the writing of this guide, Windows containers are supported only for the Amazon EC2 launch type. The Fargate launch type isn't currently supported for Windows containers. For more information, see Amazon EC2 Windows containers in the Amazon ECS documentation.

Does Amazon API Gateway support Windows-integrated authentication? Does it support OAuth 2.0?

Not by default. However, API Gateway supports OAuth 2.0 through the use of Amazon Cognito user pools, and it supports other authentication and authorization providers through the use of custom authorizers. To get started with these options, see the following:

- Introducing custom authorizers in Amazon API Gateway (AWS Compute blog)
- Control access to a REST API using Amazon Cognito user pools as authorizer (API Gateway documentation)
Next steps and resources

To begin the migration and modernization process for your legacy ASP.NET applications:

1. Determine the type of ASP.NET web services you will be migrating to AWS. From a modernization perspective, ASP.NET web services can be classified in three categories. The first two categories, discussed in this guide, are SOAP-based ASP.NET web services written either with ASMX files or by using Windows Communication Foundation (WCF), and REST-based ASP.NET services written by using RESTful ASP.NET Web APIs or the ASP.NET Model-View-Controller (MVC) pattern. The third type of ASP.NET web services are written using ASP.NET MVC and .NET Core or .NET 5. This is the current, recommended way to write ASP.NET REST APIs and should not require significant modernization to migrate to AWS.

2. Determine how these services are secured for authentication and authorization, establish your approach, and prioritize this aspect of the modernization effort.

3. Familiarize yourself with Windows containers, Amazon ECS, and Amazon API Gateway.

4. Read up on domain-driven design. This will help you make the most of your modernization efforts by ensuring that the modernized services are modeled around cohesive functional domains instead of the design of the legacy system.

5. Establish your target state architecture, and then work backward to determine the interim-state architectures needed to achieve that outcome.

For more information about the patterns discussed in this guide, see the following articles by Martin Fowler:

- Strangler fig application
- Branch by abstraction
- Domain-driven design

For more information about the AWS services discussed in this guide, see the following references:

- Amazon Elastic Container Service (Amazon ECS)
- Amazon API Gateway
AWS Prescriptive Guidance glossary

AI and ML terms (p. 12)  |  Migration terms (p. 13)  |  Modernization terms (p. 17)

AI and ML terms

The following are commonly used terms in artificial intelligence (AI) and machine learning (ML)-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.

binary classification
A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?” or “Is this product a book or a car?”

classification
A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

data preprocessing
To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

deep ensemble
To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning
An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

exploratory data analysis (EDA)
The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

features
The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance
How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.
feature transformation  
To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the “2021-05-27 00:15:37” date into “2021”, “May”, “Thu”, and “15”, you can help the learning algorithm learn nuanced patterns associated with different data components.

interpretability  
A characteristic of a machine learning model that describes the degree to which a human can understand how the model’s predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

multiclass classification  
A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask “Is this product a book, car, or phone?” or “Which product category is most interesting to this customer?”

regression  
An ML technique that predicts a numeric value. For example, to solve the problem of “What price will this house sell for?” an ML model could use a linear regression model to predict a house’s sale price based on known facts about the house (for example, the square footage).

training  
To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don’t know the target.

target variable  
The value that you are trying to predict in supervised ML. This is also referred to as an outcome variable. For example, in a manufacturing setting the target variable could be a product defect.

tuning  
To change aspects of your training process to improve the ML model’s accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

uncertainty  
A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: Epistemic uncertainty is caused by limited, incomplete data, whereas aleatoric uncertainty is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

Migration terms

The following are commonly used terms in migration-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
AWS Prescriptive Guidance Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway

- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. This migration scenario is specific to VMware Cloud on AWS, which supports virtual machine (VM) compatibility and workload portability between your on-premises environment and AWS. You can use the VMware Cloud Foundation technologies from your on-premises data centers when you migrate your infrastructure to VMware Cloud on AWS. Example: Relocate the hypervisor hosting your Oracle database to VMware Cloud on AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there’s no business justification for migrating them.
- Retire – Decommission or remove applications that are no longer needed in your source environment.

**application portfolio**
A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

**artificial intelligence operations (AIOps)**
The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

**AWS Cloud Adoption Framework (AWS CAF)**
A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

**AWS landing zone**
A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

**AWS Workload Qualification Framework (AWS WQF)**
A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Prescriptive Guidance Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway</td>
<td>Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.</td>
</tr>
<tr>
<td>Business Continuity Planning (BCP)</td>
<td>A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.</td>
</tr>
<tr>
<td>Cloud Center of Excellence (CCoE)</td>
<td>A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.</td>
</tr>
<tr>
<td>Cloud Stages of Adoption</td>
<td>The four phases that organizations typically go through when they migrate to the AWS Cloud:</td>
</tr>
<tr>
<td>• Project – Running a few cloud-related projects for proof of concept and learning purposes</td>
<td></td>
</tr>
<tr>
<td>• Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)</td>
<td></td>
</tr>
<tr>
<td>• Migration – Migrating individual applications</td>
<td></td>
</tr>
<tr>
<td>• Re-invention – Optimizing products and services, and innovating in the cloud</td>
<td></td>
</tr>
<tr>
<td>These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First &amp; the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.</td>
<td></td>
</tr>
<tr>
<td>Configuration Management Database (CMDB)</td>
<td>A database that contains information about a company's hardware and software products, configurations, and inter-dependencies. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.</td>
</tr>
<tr>
<td>Epic</td>
<td>In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.</td>
</tr>
<tr>
<td>Heterogeneous Database Migration</td>
<td>Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.</td>
</tr>
<tr>
<td>Homogeneous Database Migration</td>
<td>Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.</td>
</tr>
<tr>
<td>Idle Application</td>
<td>An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.</td>
</tr>
<tr>
<td>IT Information Library (ITIL)</td>
<td>A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>IT service management (ITSM)</td>
<td>Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.</td>
</tr>
<tr>
<td>large migration</td>
<td>A migration of 300 or more servers.</td>
</tr>
<tr>
<td>Migration Acceleration Program (MAP)</td>
<td>An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.</td>
</tr>
<tr>
<td>Migration Portfolio Assessment (MPA)</td>
<td>An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.</td>
</tr>
<tr>
<td>Migration Readiness Assessment (MRA)</td>
<td>The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.</td>
</tr>
<tr>
<td>migration at scale</td>
<td>The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a migration factory of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.</td>
</tr>
<tr>
<td>migration factory</td>
<td>Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the CloudEndure Migration Factory guide in this content set.</td>
</tr>
<tr>
<td>migration metadata</td>
<td>The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.</td>
</tr>
<tr>
<td>migration pattern</td>
<td>A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.</td>
</tr>
<tr>
<td>migration strategy</td>
<td>The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs (p. 13) entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.</td>
</tr>
<tr>
<td>operational-level agreement (OLA)</td>
<td>An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).</td>
</tr>
<tr>
<td>operations integration (OI)</td>
<td>The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.</td>
</tr>
</tbody>
</table>
organizational change management (OCM) | A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called people acceleration, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

playbook | A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

portfolio assessment | A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

responsible, accountable, consulted, informed (RACI) matrix | A matrix that defines and assigns roles and responsibilities in a project. For example, you can create a RACI to define security control ownership or to identify roles and responsibilities for specific tasks in a migration project.

runbook | A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

service-level agreement (SLA) | An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

task list | A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

workstream | Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

zombie application | An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.

**Modernization terms**

The following are commonly used terms in modernization-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.

**business capability** | What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

**domain-driven design** | An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley 17
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>microservice</td>
<td>A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.</td>
</tr>
<tr>
<td>microservices architecture</td>
<td>An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.</td>
</tr>
<tr>
<td>modernization</td>
<td>Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.</td>
</tr>
<tr>
<td>modernization readiness assessment</td>
<td>An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.</td>
</tr>
<tr>
<td>monolithic applications (monoliths)</td>
<td>Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.</td>
</tr>
<tr>
<td>polyglot persistence</td>
<td>Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.</td>
</tr>
<tr>
<td>split-and-seed model</td>
<td>A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.</td>
</tr>
<tr>
<td>strangler fig pattern</td>
<td>An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an</td>
</tr>
<tr>
<td>two-pizza team</td>
<td>example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>A small DevOps team that you can feed with two pizzas. A two-pizza team size</td>
<td>ensures the best possible opportunity for collaboration in software development. For more information, see the Two-pizza team section of the Introduction to DevOps on AWS whitepaper.</td>
</tr>
</tbody>
</table>
Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial publication (p. 20)</td>
<td>—</td>
<td>November 15, 2021</td>
</tr>
</tbody>
</table>